A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-in On-the-Fly Call Graph Construction

DONGJIE HE*, JINGBO LU*, and JINGLING XUE, UNSW Sydney, AU

For object-oriented languages, the traditional CFL-reachability formulation for k-callsite-sensitive pointer analysis (kCFA) models field accesses and calling contexts only, but relies on a separate algorithm for performing call graph construction. This may cause kCFA to lose precision even if it uses the most precise call graph for a program, built in advance or on the fly. In addition, any analysis that reasons about CFL-reachability (based on this formulation) may make optimization decisions that reduce the precision of kCFA (among others) as it is disconnected to the value-flow paths traversed by the call graph construction algorithm. As the primary contribution of this work, we overcome these two limitations by presenting the first CFL-reachability formulation of kCFA for Java with built-in on-the-fly call graph construction (as part of the same CFL-reachability formulation). As the secondary contribution of this work, we demonstrate its utility by presenting the first precision-preserving pre-analysis for accelerating kCFA with selective context-sensitivity.

CCS Concepts: • Theory of computation → Program analysis.

Additional Key Words and Phrases: Pointer Analysis, CFL Reachability, Call Graph Construction

ACM Reference Format:
Dongjie He, Jingbo Lu, and Jingling Xue. 2018. A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-in On-the-Fly Call Graph Construction. J. ACM 37, 4, Article 111 (August 2018), 42 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Pointer analysis underpins almost all forms of other static analyses. Some representative applications include program understanding, program verification, bug detection, security analysis, compiler optimization, and symbolic execution. For object-oriented programs, context-sensitive pointer analyses are the most common class of precise pointer analyses [20, 28, 52, 57]. Broadly speaking, there are two representative abstractions for context-sensitivity: (1) k-callsite-sensitivity [51], which distinguishes the contexts of a method by its k-most-recent callsites, and (2) k-object-sensitivity [40, 41], which distinguishes the contexts of a method by its receiver's k-most-recent allocation sites. In the past two decades, both abstractions have been widely used in whole-program pointer analyses [5, 42, 61], with object-sensitivity being often preferred over callsite-sensitivity in terms of the precision/-efficiency tradeoff obtained. However, a recent study [21] suggests that the converse can be true when the traditional k-limiting approach is relaxed. In the case of demand-driven

*Both authors contributed equally to this research.

Authors' address: Dongjie He, dongjieh@cse.unsw.edu.au; Jingbo Lu, jlu@cse.unsw.edu.au; Jingling Xue, jingling@cse.unsw.edu.au, UNSW Sydney, AU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

111

pointer analyses, however, callsite-sensitivity has been exclusively used [50, 54, 63] without k-limiting since on-demand points-to queries can be answered subject to a per-query time budget.

In this paper, we introduce the first CFL-reachability formulation of callsite-sensitive pointer analysis for object-oriented languages. Here, CFL stands for context-free language. We restrict ourselves to Java, since the underlying basic principle applies also to other object-oriented languages.

Traditionally, k-callsite-sensitive pointer analysis (abbreviated to kCFA (Control-Flow Analysis) [51]) for Java is either inclusion-based [1] or founded on CFL reachability [46].

On one hand, Andersen-style inclusion-based formulation for kCFA [23, 59] has been adopted in several pointer analysis frameworks for Java [5, 18, 42, 60, 61]. Given a program, its statements are modeled as points-to set constraints, its methods' calling contexts (abstracted by their last k callsites) are tracked by parameterizing these constraints with context abstractions, and its call graph is often constructed on the fly in order to achieve the best precision and efficiency possible [12, 28, 29, 49, 52].

On the other hand, the CFL-reachability formulation for kCFA [54] has also been used extensively in understanding and developing a wide range of pointer analysis algorithms, such as demand-driven pointer/alias analysis [50, 54, 55, 63, 65], context transformations [59], specification inference [3], library-code summarization [50, 58], information flow analysis [32, 39], and selective context-sensitivity [30, 36]. Given a program, its points-to information is computed by solving a graph reachability problem over a so-called pointer assignment graph (PAG) [28]. Such a CFL-reachability formulation consists of reasoning about the intersection of two CFLs, $L_{FC} = L_F \cap L_C$, where L_F describes field accesses as balanced parentheses and L_C enforces callsite-sensitivity by matching method calls and returns as also balanced parentheses [54]. However, a separate algorithm, which is formulated outside L_{FC}, is used for call graph construction (as described in Sec. 2).

Compared with Andersen-style inclusion-based formulation, this L_{FC}-based CFL-reachability formulation for specifying kCFA suffers from two major limitations due to the lack of a built-in call graph construction mechanism. First, kCFA may lose precision even if it uses the most precise call graph for a program (built in advance or on the fly). Second, any analysis that reasons about CFL-reachability in terms of L_{FC} will fail to make a connection with the value-flow paths traversed by a separate call graph construction algorithm used, and consequently, may make optimization decisions that reduce the precision of kCFA (among others).

In this paper, as the primary contribution of this research, we overcome these two limitations by introducing a CFL-reachability formulation of kCFA for Java with built-in on-the-fly call graph construction (as part of the same CFL-reachability formulation). Adapting the previous formulation used in object-sensitive pointer analysis [35, 37] to our approach poses significant challenges and complexities, as discussed in Section 3.2.1.2 and further explored in Section 5. Furthermore, an earlier attempt to address the same problem, presented in Sridharan's PhD thesis [53], falls short in providing a comprehensive solution, as will be discussed in detail in Section 5. Our formulation consists of reasoning about the intersection of three CFLs, $L_{DCR} = L_D \cap L_C \cap L_R$, over a new PAG representation for a Java program, where L_D specifies not only field accesses as in L_F but also dynamic method dispatch, L_C enforces callsite-sensitivity exactly as before [54], and L_R supports parameter passing in the presence of built-in on-the-fly call graph construction. Theoretically, we present a novel insight by demonstrating for the first time that callsite-sensitive pointer analysis can be formulated as a particular type of context-sensitive language. Specifically, we express it as the intersection of multiple CFLs. It is important to note that not all context-sensitive languages can be expressed in this manner[27, 34], highlighting the uniqueness of our approach. We will discuss several challenges faced in designing L_{DCR} and provide some insights for understanding our formulation. Note that the Melski-Reps reduction [38] cannot be used to convert Andersen-style inclusion-based formulation for kCFA into L_{DCR} since L_{DCR} is the intersection of three different CFLs rather than just one single CFL.

L_{DCR} can be applied to all the applications that benefit from L_{FC}. By formulating kCFA (with built-in on-the-fly callgraph construction) in terms of CFL-reachability, L_{DCR} can be potentially more useful than L_{FC} for several recently-studied CFL-reachability-based analyses: specification inference [3], library-code summarization [50, 58], information flow analysis [32, 39], and selective context-sensitivity [30, 36]. To demonstrate its utility, as the secondary contribution of this research, we introduce the first L_{DCR}-enabled precision-preserving pre-analysis for accelerating kCFA for Java with selective context-sensitivity, which also serves to validate the correctness of L_{DCR}. In contrast, a recently proposed pre-analysis [36] will always lose precision as it is developed based on L_{FC} [54].

In summary, this paper makes the following two contributions:

- Our primary contribution is to introduce a CFL-reachability formulation L_{DCR} of kCFA for object-oriented languages with on-the-fly call graph construction being built into the formulation itself (rather than done by a separate call graph construction algorithm), demonstrating for the first time that callsite-sensitive pointer analysis represents a special kind of context-sensitive language that can be expressed by the intersections of several CFLs.
- Our secondary contribution is to introduce an L_{DCR}-enabled precision-preserving pre-analysis for accelerating kCFA for object-oriented languages with selective context-sensitivity. Compared with two state-of-the-art pre-analyses [30, 36], our pre-analysis enables better efficiency-precision trade-offs to be made in several application scenarios outlined.

The rest of this paper is organized as follows. Sec. 2 provides some background knowledge and motivates the development of L_{DCR} by highlighting several challenges faced in its design. Sec. 3 introduces L_{DCR} by explaining how we address these challenges and providing some insights in understanding its design. Sec. 4 introduces and evaluates our L_{DCR}-enabled pre-analysis for accelerating kCFA. Sec. 5 discusses the related work. Sec. 6 concludes the paper.

2 Background and Motivation

This paper focuses on kCFA for object-oriented languages such as Java. For a Java program, its call graph is constructed by discovering the target methods invoked at its virtual callsites. For kCFA, we first review its Andersen-style inclusion-based formulation and its traditional CFL-reachability formulation L_{FC} (Sec. 2.1). We then motivate this research with an example, by illustrating how these two formulations handle call graph construction, examining the limitations of L_{FC}, and finally, highlighting the necessity of and challenges faced in designing L_{DCR}, a new CFL-reachability formulation that includes on-the-fly call graph construction as a built-in mechanism (Sec. 2.2).

We consider six types of statements given in Table 1, where `x` and `y` are local variables, `O` represents a unique abstract object created by a `new` statement, and `c` identifies a callsite. By nature, all global variables are always context-insensitive. Without loss of generality, every method is assumed to have a unique return statement "return v", where v is a local variable referred to as its return variable. Given a virtual call $\mathtt{r.m}(a_1, \ldots, a_n)$, we write $\mathtt{this}^{\mathtt{m}'}$,

Kind	Statement	Kind	Statement
New	`x = new T // O`	Assign	`x = y`
Store	`x.f = y`	Load	`x = y.f`
Virtual Call	x = r.m($a_1, \ldots, a_n$) // c	Static Call	x = m($a_1, \ldots, a_n$) // c

Table 1. Six types of statements.

$p_i^{\mathtt{m}'}$ and $\mathtt{ret}^{\mathtt{m}'}$ as the "this" variable, i-th parameter and return variable of a virtual method $\mathtt{m}'$ invoked at this particular callsite. For a static call $\mathtt{m}(a_1, \ldots, a_n)$, $p_i^{\mathtt{m}}$ and $\mathtt{ret}^{\mathtt{m}}$ are used instead.

2.1 Background

2.1.1 Andersen-Style Inclusion-based Formulation In the Andersen-style [1] formulation [23, 59] given in Fig. 1, several auxiliary functions are used: (1) MethodCtx maintains the contexts used for analyzing a method, (2) dispatch follows Java's standard single-dispatch semantics by resolving a virtual call to a target method according to the type of its receiver object, and (3) PTS records the points-to information found context-sensitively for a variable or an object's field. In kCFA, a calling context of a method is abstracted by its last k callsites (under k-limiting). Context-sensitivity is achieved by parameterizing variables and objects with contexts as modifiers.

Let us examine the six rules in Fig. 1. Given a context $ctx = [c_1, \ldots, c_n]$ and a context element c, we write $c :: ctx$ to represent $[c, c_1, \ldots, c_n]$ and $\lceil ctx \rceil_k$ to represent $[c_1, \ldots, c_k]$. In [I-New], hk represents the (heap) context length for a heap object. In practice, $hk = k - 1$ is usually used [20, 31, 57]. Rules [I-Assign], [I-Load], and [I-Store] handle assignments and field accesses in the standard manner. [I-SCall] and [I-VCall] handle static and virtual calls, respectively. Let us explain [I-VCall] only, as [I-SCall] can be understood similarly. In this rule, $\mathtt{m}'$ is a target method dynamically resolved for a receiver object O at callsite c (based on its dynamic type $\mathtt{t} = \mathtt{DynTypeOf}(O)$). Thus, this rule is also responsible for performing on-the-fly call graph construction during the pointer analysis. In its conclusion, $ctx' \in \mathtt{MethodCtx}(\mathtt{m}')$ reveals how the method contexts of a method are introduced. Initially, the entry methods in a program have only the empty context, e.g., MethodCtx("main") = {[ ]}. Note that the receiver variable r and the other arguments $a_1, \ldots, a_n$ are handled differently. A receiver object flows only to the method dispatched on itself while the objects pointed to by the other arguments flow to all the methods dispatched at this callsite.

2.1.2 L_{FC}-based CFL-Reachability Formulation In L_{FC} [54], kCFA for a program is solved by reasoning about CFL-reachability on a PAG (Pointer Assignment Graph) representation [28]. Fig. 2 gives six rules for building the PAG. For a PAG edge, its label above indicates whether it is an assignment or field access. There are two types of assign edges: intra-procedural edges (for modeling regular assignments without a below-edge label) and inter-procedural edges or call edges (for modeling parameter passing with a below-edge label representing a callsite).

In L_{FC}, passing arguments to parameters at both static and virtual callsites is modeled identically by inter-procedural assign edges ([P-SCall] and [P-VCall]). For example, in [P-VCall], $\hat{c}$ ($\check{c}$) signifies an inter-procedural value-flow entering into (exiting from) $\mathtt{m}'$ at callsite c, where $\mathtt{m}'$ represents a virtual method discovered by a separate call graph construction

$$\frac{\texttt{x = }\textbf{new}\texttt{ T // O} \quad ctx \in \texttt{MethodCtx(M)} \quad htx = \lceil ctx \rceil_{hk}}{\langle O, htx \rangle \in \mathsf{PTS}(\texttt{x}, ctx)} \qquad \text{[I-NEW]}$$

$$\frac{\texttt{x = y} \quad ctx \in \texttt{MethodCtx(M)}}{\mathsf{PTS}(\texttt{y}, ctx) \subseteq \mathsf{PTS}(\texttt{x}, ctx)} \qquad \text{[I-ASSIGN]}$$

$$\frac{\texttt{x = y.f} \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx \rangle \in \mathsf{PTS}(\texttt{y}, ctx)}{\mathsf{PTS}(O.\mathsf{f}, htx) \subseteq \mathsf{PTS}(\texttt{x}, ctx)} \qquad \text{[I-LOAD]}$$

$$\frac{\texttt{x.f = y} \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx \rangle \in \mathsf{PTS}(\texttt{x}, ctx)}{\mathsf{PTS}(\texttt{y}, ctx) \subseteq \mathsf{PTS}(O.\mathsf{f}, htx)} \qquad \text{[I-STORE]}$$

$$\frac{\texttt{x = m}(a_1, \ldots, a_n)\texttt{ // c} \quad ctx \in \texttt{MethodCtx(M)} \quad ctx' = \lceil \texttt{c} :: ctx \rceil_k}{\begin{array}{c} ctx' \in \texttt{MethodCtx(m)} \quad \mathsf{PTS}(\texttt{ret}^{\texttt{m}}, ctx') \subseteq \mathsf{PTS}(\texttt{x}, ctx) \\ \forall i \in [1, n] : \mathsf{PTS}(a_i, ctx) \subseteq \mathsf{PTS}(p_i^{\texttt{m}}, ctx') \end{array}} \qquad \text{[I-SCALL]}$$

$$\frac{\begin{array}{c} \texttt{x = r.m}(a_1, \ldots, a_n)\texttt{ // c} \quad ctx \in \texttt{MethodCtx(M)} \quad \langle O, htx \rangle \in \mathsf{PTS}(\texttt{r}, ctx) \\ \texttt{t} = \texttt{DynTypeOf}(O) \quad \texttt{m}' = \texttt{dispatch(m, t)} \quad ctx' = \lceil \texttt{c} :: ctx \rceil_k \end{array}}{\begin{array}{c} ctx' \in \texttt{MethodCtx(m}'\texttt{)} \quad \mathsf{PTS}(\texttt{ret}^{\texttt{m}'}, ctx') \subseteq \mathsf{PTS}(\texttt{x}, ctx) \\ \langle O, htx \rangle \in \mathsf{PTS}(\texttt{this}^{\texttt{m}'}, ctx') \quad \forall i \in [1, n] : \mathsf{PTS}(a_i, ctx) \subseteq \mathsf{PTS}(p_i^{\texttt{m}'}, ctx') \end{array}} \qquad \text{[I-VCALL]}$$

Fig. 1. Andersen-style inclusion-based formulation (where `M` is the containing method of a statement being analyzed). $\langle O, htx \rangle$ is used to denote a pair of values, where htx is the (heap) context of O.

algorithm (either in advance [2, 9, 56] or on the fly [54, 55]). Therefore, $\hat{c}$ ($\check{c}$) is also known as an entry (exit) context.

For a PAG edge $x \xrightarrow[c]{\ell} y$, its inverse edge, which is omitted in Fig. 2 but required by L_{FC}, is defined as $y \xrightarrow[\bar{c}]{\bar{\ell}} x$. For a below-edge context label $\hat{c}$ or $\check{c}$, $\bar{\hat{c}} = \check{c}$ and $\bar{\check{c}} = \hat{c}$, implying that the concepts of entry and exit contexts for inter-procedural assign edges are swapped if they are traversed inversely.

In this CFL-reachability formulation, kCFA is solved by reasoning about the intersection of two CFLs, $L_{FC} = L_F \cap L_C$, where L_F enforces field-sensitivity in terms of the PAG's above-edge labels and L_C enforces context-sensitivity in terms of the PAG's below-edge labels. Unlike the earlier work [50, 54, 55, 63, 65], which uses one label per PAG edge, our work allows a two-label PAG edge of the form of $x \xrightarrow[\ell_b]{\ell_a} y$, which can be understood as a shorthand for a sequence of two single-label edges $x \xrightarrow{\ell_a} t \xrightarrow{\ell_b} y$, where t is a fresh node, in order to simplify our presentation. Let L be a CFL over Σ formed by all the edge labels in a given PAG. Each path p in the PAG has a string $L(p)$ in Σ^* formed by concatenating in order the edge labels in p. A node v in the PAG is said to be L-reachable from a node u in the PAG if there exists a path p from u to v, known as L-path, such that $L(p) \in L$.

$$\frac{\texttt{x = }\textbf{new}\texttt{ T // O}}{\texttt{O} \xrightarrow{\text{new}} \texttt{x}}\ [\text{P-New}] \qquad \frac{\texttt{x = y}}{\texttt{y} \xrightarrow{\text{assign}} \texttt{x}}\ [\text{P-Assign}]$$

$$\frac{\texttt{x = y.f}}{\texttt{y} \xrightarrow{\text{load[f]}} \texttt{x}}\ [\text{P-Load}] \qquad \frac{\texttt{x.f = y}}{\texttt{y} \xrightarrow{\text{store[f]}} \texttt{x}}\ [\text{P-Store}]$$

$$\frac{\texttt{x = m}(a_1, \ldots, a_n)\ \texttt{// c}}{\forall\, i \in [1, n] : a_i \xrightarrow[\hat{c}]{\text{assign}} p_i^{\texttt{m}} \quad \texttt{ret}^{\texttt{m}} \xrightarrow[\check{c}]{\text{assign}} \texttt{x}}\ [\text{P-SCall}]$$

$$\frac{\texttt{x = r.m}(a_1, \ldots, a_n)\ \texttt{// c} \quad \texttt{m}' \text{ is a target method of this callsite}}{\texttt{r} \xrightarrow[\hat{c}]{\text{assign}} \texttt{this}^{\texttt{m}'} \quad \forall\, i \in [1, n] : a_i \xrightarrow[\hat{c}]{\text{assign}} p_i^{\texttt{m}'} \quad \texttt{ret}^{\texttt{m}'} \xrightarrow[\check{c}]{\text{assign}} \texttt{x}}\ [\text{P-VCall}]$$

Fig. 2. Rules for building the PAG required by L_{FC}.

We give L_F and L_C below and illustrate both CFLs with an example in Sec. 2.2. We express a grammar for defining a CFL in BackusNaur form. L_F (L_C) is defined to enforce field-sensitivity (context-sensitivity) by focusing on reasoning about above-edge (below-edge) labels exclusively. When giving its grammar, we will follow existing CFL-reachability formulations [50, 54, 63] to list explicitly only the set of productions involving above-edge labels (below-edge) labels, but discuss how below-edge (above-edge) labels can be modeled easily by the set of productions omitted.

L_F enforces field-sensitivity by matching stores and loads as balanced parentheses:

$$\begin{array}{rcl}
\text{flowsto} & \longrightarrow & \text{new flows}^* \\
\text{flows} & \longrightarrow & \text{assign} \mid \text{store[f] alias load[f]} \\
\text{alias} & \longrightarrow & \overline{\text{flowsto}}\ \text{flowsto} \\
\overline{\text{flowsto}} & \longrightarrow & \overline{\text{flows}}^*\ \overline{\text{new}} \\
\overline{\text{flows}} & \longrightarrow & \overline{\text{assign}} \mid \overline{\text{load[f]}}\ \text{alias}\ \overline{\text{store[f]}}
\end{array} \tag{1}$$

where only the above-edge labels in a PAG are mentioned explicitly. It is understood that all the below-edge (context) labels that are not mentioned explicitly are handled implicitly as follows. As discussed above, each inter-procedural assign edge $x \xrightarrow[c]{\text{assign}} y$, where $c \in \{\hat{c}, \check{c}\}$, is modeled as a sequence of two single-label edges $x \xrightarrow{\text{assign}} t \xrightarrow{c} y$, where t is a fresh node. As a result, its inverse has been similarly decomposed into two single-label edges. Afterwards, flows is extended to become flows $\longrightarrow$ ... | c and $\overline{\text{flows}}$ is similarly extended to become $\overline{\text{flows}}$ $\longrightarrow$ ... | $\bar{c}$.

Note that u alias v iff u $\overline{\text{flowsto}}$ O flowsto v for some object O. If O flowsto v (via assignments or store-load pairs with aliased base variables), then v is L_F-reachable from O. In addition, O flowsto v iff v $\overline{\text{flowsto}}$ O, meaning that $\overline{\text{flowsto}}$ actually represents the standard points-to relation.

L_C enforces callsite-sensitivity (by matching “calls” and “returns” as also balanced parentheses):

$$\begin{array}{rcl} \mathsf{realizable} & \longrightarrow & \mathsf{exit\ entry} \\ \mathsf{exit} & \longrightarrow & \mathsf{exit\ balanced} \mid \mathsf{exit}\ \check{c} \mid \epsilon \\ \mathsf{entry} & \longrightarrow & \mathsf{entry\ balanced} \mid \mathsf{entry}\ \hat{c} \mid \epsilon \\ \mathsf{balanced} & \longrightarrow & \mathsf{balanced\ balanced} \mid \hat{c}\ \mathsf{balanced}\ \check{c} \mid \epsilon \end{array} \tag{2}$$

where only the below-edge (context) labels in a PAG are mentioned explicitly. To accommodate all the above-edge labels explicitly, each inter-procedural assign edge is decomposed into two single-label edges as done when L_F is introduced above. Afterwards, balanced can be extended by adding a new production balanced $\longrightarrow$ no-call-edge-label, where the new non-terminal non-call-edge-label is defined in terms of all the other non-context edge labels in a PAG [53].

A path p in the PAG is said to be realizable if and only if p is an L_C-path.

Finally, a variable v points to an object O if and only if there exists a path p (referred to as an L_{FC}-path) from O to v in the PAG, such that $L_F(p) \in L_F$ (indicating that p is a flowsto-path) and $L_C(p) \in L_C$ (indicating that p is a realizable-path). With all balanced contexts ignored, the contexts for v and O can be directly read off from p (as described in Sec. 3.2.2).

2.2 Motivation

To motivate our work, we use a small program (Sec. 2.2.1). We start with the Andersen-style inclusion-based formulation that comes with its own on-the-fly call graph construction mechanism (Sec. 2.2.2). We then examine the limitations of L_{FC} when such a built-in mechanism is absent (Sec. 2.2.3). Finally, we discuss several challenges faced in designing our new CFL-reachability formulation, L_{DCR}, with on-the-fly call graph construction being built-in (Sec. 2.2.4). When moving from L_{FC} to L_{DCR}, we also rely on a new PAG representation for a program for L_{DCR} to operate on.

2.2.1 Example Consider a Java program given in Fig. 3. Given a class `T`, we write `T:foo()` for method `foo()` defined in `T`. There are five classes, `A`, `B`, `C`, `D` and `O`, defined (lines 1-13). `B` and `C` are the subclasses of `A`, both overriding method `foo()` defined in `A`. Method `bar()` (lines 14-18) is a wrapper method which first stores whatever object pointed by its parameter `o` into `D1.f` and then invokes `A:foo()` or `B:foo()`, depending on the dynamic type of the object pointed by its parameter `x`. In `main()`, four objects, `O1`, `O2`, `A1` and `B1`, are created, in which `A1` and `O1` (`B1` and `O2`) are passed into `bar()` as its first and second arguments, respectively, at callsite `c1` (`c2`).

Note that `C:foo()` may be regarded as being called conservatively in line 17 by a pointer analysis algorithm even though this can never happen during program execution. At the end of Sec. 3.2.2, we will see how our CFL-reachability formulation L_{DCR} avoid analyzing such a spurious call.

2.2.2 Andersen-Style Inclusion-based Formulation According to Fig. 1, [I-VCall] not only discovers dynamically the target methods dispatched at a virtual callsite but also propagates iteratively the points-to information inter-procedurally across the call graph thus built on the fly.

Table 2 lists the points-to results computed for the program in Fig. 3 by 2CFA according to the rules in Fig. 1. For `main()`, analyzed under [ ], its points-to relations are obtained trivially.

```
 1 class A {                      14 static void bar(A x, O o) {
 2   void foo(D p) {              15   D d = new D(); // D1
 3     Object v = p.f;            16   d.f = o;
 4   }                            17   x.foo(d); // c3
 5 }                              18 }
 6 class B extends A {            19 static void main() {
 7   void foo(D q) { }            20   O o1 = new O(); // O1
 8 }                              21   O o2 = new O(); // O2
 9 class C extends A {            22   A a = new A(); // A1
10   void foo(D r) { }            23   A b = new B(); // B1
11 }                              24   bar(a, o1); // c1
12 class D { Object f; }          25   bar(b, o2); // c2
13 class O { }                    26 }
```

Fig. 3. A motivating example.

Table 2. The points-to results for the program in Fig. 3 computed by 2CFA according to the rules in Fig. 1.

Method	Pointers	PTS	Method	Pointers	PTS
main()	⟨o1, [ ]⟩	{⟨O1, [ ]⟩}	bar()	⟨x, [c1]⟩	{⟨A1, [ ]⟩}
	⟨o2, [ ]⟩	{⟨O2, [ ]⟩}		⟨o, [c1]⟩	{⟨O1, [ ]⟩}
	⟨a, [ ]⟩	{⟨A1, [ ]⟩}		⟨d, [c1]⟩	{⟨D1, [c1]⟩}
	⟨b, [ ]⟩	{⟨B1, [ ]⟩}		⟨x, [c2]⟩	{⟨B1, [ ]⟩}
A:foo()	⟨this, [c3, c1]⟩	{⟨A1, [ ]⟩}		⟨o, [c2]⟩	{⟨O2, [ ]⟩}
	⟨p, [c3, c1]⟩	{⟨D1, [c1]⟩}		⟨d, [c2]⟩	{⟨D1, [c2]⟩}
	⟨v, [c3, c1]⟩	{⟨O1, [ ]⟩}	Field	Pointers	PTS
B:foo()	⟨this, [c3, c2]⟩	{⟨B1, [ ]⟩}	f	⟨D1.f, [c1]⟩	{⟨O1, [ ]⟩}
	⟨q, [c3, c2]⟩	{⟨D1, [c2]⟩}		⟨D1.f, [c2]⟩	{⟨O2, [ ]⟩}

As for `bar()`, there are two calling contexts, [c1] and [c2]. Under [c1], we have $\mathsf{PTS}(x, [c1]) = \{\langle A1, [\,]\rangle\}$, $\mathsf{PTS}(d, [c1]) = \{\langle D1, c1\rangle\}$, and $\mathsf{PTS}(D1.f, [c1]) = \mathsf{PTS}(o, [c1]) = \{\langle O1, [\,]\rangle\}$. Then `A:foo()` is found to be the target invoked by `x.foo()` at callsite `c3` in line 17 ([I-VCall]). Thus, $\mathsf{PTS}(p, [c3, c1]) = \{\langle D1, [c1]\rangle\}$ and $\mathsf{PTS}(v, [c3, c1]) = \{\langle O1, [\,]\rangle\}$. Similarly, when `bar()` is analyzed under [c2], we have $\mathsf{PTS}(x, [c2]) = \{\langle B1, [\,]\rangle\}$. Thus, `x.foo()` at callsite `c3` is now resolved to `B:foo()`. Note that 2CFA is precise enough by not resolving `C:foo()` as a spurious target at callsite `c3`.

2.2.3 *L_{FC}-based CFL-Reachability Formulation* In this traditional L_{FC}-based framework for solving kCFA [54], a separate algorithm for call graph construction is used. Thus, for a virtual callsite, parameter passing that is prescribed by L_{FC} is disconnected both conceptually and algorithmically with the dynamic dispatch process done at the callsite. We discuss

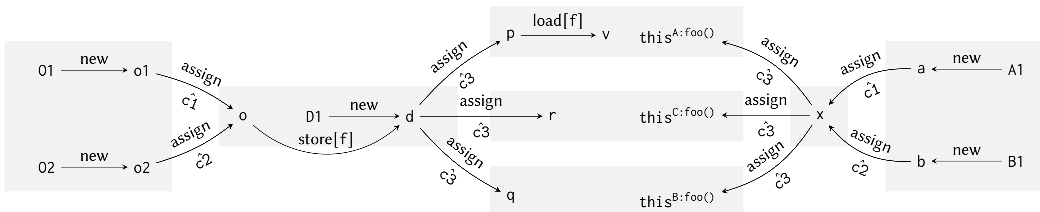


Fig. 4. The PAG operated on by L_{FC} for the program given in Fig. 3.

the resulting limitations by considering whether the call graph is constructed in advance and on the fly.

For the program given in Fig. 3, L_{FC} operates on its PAG shown in Fig. 4. This PAG is built by using Class Hierarchy Analysis (CHA) [9], in which case, `C:foo()` is also identified conservatively as a target method at callsite `c3` (line 17). As will be clear below, L_{FC} will filter out such a spurious target if it uses a more precise call graph during its actual analysis.

We consider a particular traversal just reaching `d`, which is an argument in the call at "`x.foo(d); // c3`" (line 17), starting originally from either `O1` when called from `bar(a,o1)` under [`c1`] or `O2` when called from `bar(b,o2)` under [`c2`]. We now need to pass `d` to its corresponding parameter `p` if `A:foo()` is a target, `q` if `B:foo()` is a target, and `r` if `C:foo()` is a target at this callsite.

2.2.3.1 Using a Call Graph Constructed in Advance *k*CFA may lose precision even if L_{FC} uses the most precise pre-built call graph obtained in advance. In this case, the set of methods that may possibly be invoked at "`x.foo(d); // c3`" (line 17) in Fig. 3 is found to contain both `A:foo()` and `B:foo()` independently of the contexts in which this call is made. Thus, regardless of whether the call is triggered by `bar(a,o1)` under [`c1`] or `bar(a,o2)` under [`c2`], `A:foo()` is always a target method to be invoked. As a result, due to the existence of the following two L_{FC}-paths:

$$\mathtt{O1} \xrightarrow{\mathsf{new}} \mathtt{o1} \xrightarrow[\hat{\mathtt{c1}}]{\mathsf{assign}} \mathtt{o} \xrightarrow{\mathsf{store[f]}} \mathtt{d} \xrightarrow{\overline{\mathsf{new}}} \mathtt{D1} \xrightarrow{\mathsf{new}} \mathtt{d} \xrightarrow[\hat{\mathtt{c3}}]{\mathsf{assign}} \mathtt{p} \xrightarrow{\mathsf{load[f]}} \mathtt{v} \tag{3}$$

$$\mathtt{O2} \xrightarrow{\mathsf{new}} \mathtt{o2} \xrightarrow[\hat{\mathtt{c2}}]{\mathsf{assign}} \mathtt{o} \xrightarrow{\mathsf{store[f]}} \mathtt{d} \xrightarrow{\overline{\mathsf{new}}} \mathtt{D1} \xrightarrow{\mathsf{new}} \mathtt{d} \xrightarrow[\hat{\mathtt{c3}}]{\mathsf{assign}} \mathtt{p} \xrightarrow{\mathsf{load[f]}} \mathtt{v} \tag{4}$$

this L_{FC}-based CFL-reachability pointer analysis will conclude that `v` point to both `O1` and `O2` although `v` points to `O1` only by 2CFA (Table 2), meaning that the pointed-to object `O2` is spurious.

Why is the precision loss? In L_{FC}, parameter passing for a virtual callsite ([P-VCall]) is modeled identically as that at a static callsite ([P-SCall]) by using inter-procedural `assign` edges as shown in the two L_{FC}-paths given above, without being CFL-reachability-related to the receiver objects at the callsite. As a result, L_{FC} does not really understand that under context [`c1`], in which case `x` points to the receiver `A1`, only the first L_{FC}-path above can be established.

If L_{FC} uses a less precise call graph, which is pre-built by, say, CHA [9], then `C:foo()` will also be identified as a target method at callsite `c3` (line 17). In this case, `r` is found to point to `D1` due to $\mathtt{D1} \xrightarrow{\mathsf{new}} \mathtt{d} \xrightarrow[\hat{\mathtt{c3}}]{\mathsf{assign}} \mathtt{r}$, but its points-to set is empty by 2CFA (not listed in Table 2).

2.2.3.2 Using a Call Graph Constructed On the Fly In solving kCFA with L_{FC} on-demand [50, 54, 63], every method that is invoked at a virtual callsite is dispatched only under a specific context, resulting in on-the-fly call graph construction (which implies that the PAG edges related to a call are not always fixed but provided to L_{FC} when the call is analyzed under a given context).

Consider again "`x.foo(d); // c3`" (line 17) in Fig. 3. We can now establish that the path in Eq. (3) is an L_{FC}-path but the path in Eq. (4) is not, so that we can conclude precisely that `v` points to `O1` only. In the former case, we reach `d` under context [`c1`] and then issue a points-to query to find what `x` points to under [`c1`]. As `x` is found to point to `A1` in this case (causing `A:foo()` to be invoked at callsite `c3`), we will continue traversing the remaining L_{FC}-path from `d` and conclude that `v` points to `O1`. In the latter case, reaching `d` under [`c2`] reveals `B:foo()` as the target at callsite `c3` instead (as `x` points to `B1` under [`c2`]), thereby causing $\xrightarrow[\hat{\mathsf{c3}}]{\mathsf{assign}} \mathsf{p} \xrightarrow{\mathsf{load[f]}} \mathsf{v}$ not to be traversed.

```
D d = new D(); // D1
if (...)
  d.f = a = new A(); // A1
else
  d.f = b = new B(); // B1
A x = d.f;
x.foo(null); // c
```

Fig. 5. A small example.

While L_{FC} can be used to solve kCFA on-demand (more precisely than if a pre-built call graph is used), some precision loss may occur when a callsite has several dispatch targets under a common calling context. Consider the code snippet given in Fig. 5 (which reuses classes `A`, `B`, and `D` from Fig. 3). If we ask a separate call graph construction algorithm to find on-demand the target methods at "`x.foo(`**`null`**`)`" under any context invoking this piece of code, both `A:foo()` and `B:foo()` will be returned. If we then reason about CFL-reachability with L_{FC}, we will obtain:

$$\mathsf{A1} \xrightarrow{\mathsf{new}} \mathsf{a} \xrightarrow{\mathsf{store[f]}} \mathsf{d} \xrightarrow{\overline{\mathsf{new}}} \mathsf{D1} \xrightarrow{\mathsf{new}} \mathsf{d} \xrightarrow{\mathsf{load[f]}} \mathsf{x} \xrightarrow[\hat{\mathsf{c}}]{\mathsf{assign}} \mathsf{this}^{\mathsf{A:foo()}} \tag{5}$$

$$\mathsf{B1} \xrightarrow{\mathsf{new}} \mathsf{b} \xrightarrow{\mathsf{store[f]}} \mathsf{d} \xrightarrow{\overline{\mathsf{new}}} \mathsf{D1} \xrightarrow{\mathsf{new}} \mathsf{d} \xrightarrow{\mathsf{load[f]}} \mathsf{x} \xrightarrow[\hat{\mathsf{c}}]{\mathsf{assign}} \mathsf{this}^{\mathsf{A:foo()}} \tag{6}$$

Therefore, both `A1` and `B1` will flow to $\mathsf{this}^{\mathsf{A:foo}}$ although `B1` is spurious by [I-VCall].

We see a loss of precision at such a virtual callsite since L_{FC} does not handle its receiver variable differently from its other arguments ([P-VCall]) unlike the Andersen-style inclusion-based formulation ([I-VCall]). Removing spurious receiver objects such as `B1` by brute force as discussed above (specified formally by neither L_{FC} nor the call graph construction algorithm used) is ad hoc. Indeed, the L_{FC}-based on-demand algorithm for solving kCFA (released by the authors of L_{FC} [54] in Soot [60] and used by many others [50, 62] in the past 15 years) suffers still from this problem.

2.2.3.3 Discussion When solving kCFA, L_{FC} relies on a separate algorithm for performing call graph construction. In addition to cause kCFA to lose precision as discussed above, L_{FC} suffers from another limitation, as it fails to capture all the value-flow paths traversed for a program during the pointer analysis (regardless of whether its call graph is built in advance or on the fly).

Consider again how "`x.foo(d)`" (line 17) in Fig. 3 is analyzed. To perform parameter passing for `d` at the callsite according to the Andersen-style inclusion-based formulation ([I-VCall]), we must first find the methods dispatched on the receiver objects pointed by `x` and then perform the actual parameter passing (from `d` to `p` if `A:foo()` is dispatched and `d` to `q` if `B:foo()` is dispatched). However, with L_{FC}, parameter passing (realized by inter-procedural assign edges ([P-VCall])) is both conceptually and algorithmically disconnected with dynamic dispatch at the callsite, without being CFL-reachability-related to its receiver objects, as also reviewed by the PAG in Fig. 4.

As L_{FC} does not incorporate callgraph construction within its formulation, any analysis that reasons about CFL-reachability in terms of L_{FC} may make some optimization decisions that reduce the precision of kCFA (among others). For example, a recent pre-analysis [36] that is developed based on L_{FC} for accelerating kCFA with selective context-sensitivity will cause kCFA to always lose precision.

2.2.4 L_{DCR}: Necessity, Challenges, and Our Solution Our primary contribution in this research is demonstrating the feasibility of incorporating on-the-fly callgraph construction into the specification of kCFA using CFL-reachability. In particular, we introduce L_{DCR} (as the intersection of three CFLs) as the first CFL-reachability formulation of kCFA with built-in call graph construction, thereby overcoming the limitations of L_{FC} (as the intersections of two CFLs) discussed above. It is worth emphasizing that L_{DCR} operates on a new PAG representation that is fundamentally different from the one operated on by L_{FC}. As the secondary contribution of this research, we demonstrate the utility of L_{DCR} by introducing the first precision-preserving pre-analysis for accelerating kCFA with selective context-sensitivity. As discussed above, a recent L_{FC}-based pre-analysis always loses precision.

When developing L_{DCR}, we are required to reason about CFL reachability to support parameter passing prescribed by kCFA. For a virtual call $\mathtt{r}.\mathtt{m}(a_1, \ldots, a_n)$ at callsite c, passing any of its arguments a to its corresponding parameter p of a target method m' that is yet to be discovered at this callsite by L_{DCR} itself under a given context C must be done by establishing a CFL-reachability path in a PAG representation of the program, starting from a, passing through the receiver variable `r` to trigger dynamic dispatch, and finally, ending at p, which is the corresponding parameter of m' that is dispatched based on the dynamic type of the object pointed by `r` under the given context C. Relating a to `r` is non-trivial if $a = a_i$ (for some i), i.e., $a \neq \mathtt{r}$. In addition, in a CFL-reachability formulation, some context elements in C are usually consumed (i.e., balanced out according to L_C given in Eq. (2)) during the traversal for performing dynamic dispatch and must thus be restored in order to enable `d` to be passed to `p` under still the same given context C. Below we list three challenges faced in handling this parameter passing task during the on-the-fly call graph construction:

- CHL1. How do we pass `r` to the "this" variable of a target method invoked at callsite c precisely (by avoiding the precision loss illustrated with the code given in Fig. 5)?
- CHL2. How do we establish a CFL-reachability path in a PAG representation of the program from a_i to p_i passing through `r` in order to trigger dynamic dispatch during

the course of parameter passing, where p_i is the i-th parameter of a target method m' discovered at callsite c under C?

- CHL3. How do we ensure we can pass a_i to p_i for the target method m' invoked at callsite c with a (correct) context abstraction that represents parameter passing for callsite c under C?

We now give a high-level overview of our solution by using our motivating example (Fig. 3). To support on-the-fly call graph construction by itself, L_{DCR} operates on a new PAG representation shown in Fig. 7, which differs fundamentally from the PAG (Fig. 4) operated on by L_{FC}. In our formulation, we find that `v` points to `O1` only due to the existence of the following unique path from `O1` to `v`, with the underlying technical details being explained in Sec. 3:

$$\texttt{O1} \xrightarrow{\text{new[O]}} \texttt{o1} \xrightarrow[\hat{\texttt{c1}}]{\text{assign}} \texttt{o} \xrightarrow{\text{store[f]}} \texttt{d} \xrightarrow{\overline{\text{new[D]}}} \texttt{D1} \xrightarrow{\text{new[D]}} \texttt{d} \xrightarrow[\boxed{\hat{\texttt{c3}}}]{\text{store[1]}} \texttt{x} \xrightarrow[\check{\texttt{c1}}]{\overline{\text{assign}}} \texttt{a} \xrightarrow{\overline{\text{new[A]}}} \texttt{A1}$$
$$\xrightarrow{\text{new[A]}} \texttt{a} \xrightarrow[\hat{\texttt{c1}}]{\text{assign}} \texttt{x} \xrightarrow[\boxed{\check{\texttt{c3}}}]{\text{assign}} \texttt{x\#c3} \xrightarrow[\hat{\texttt{c3}}]{\text{dispatch[A]}} \texttt{this}^{\texttt{A:foo()}} \xrightarrow{\text{load[1]}} \texttt{p} \xrightarrow{\text{load[f]}} \texttt{v} \tag{7}$$

This path represents the flow of `O1` to `v` by passing a sequence of two calls, "`bar(a,o1); // c1`" and "`x.foo(d); // c3`". Consider the parameter passing for `d` under context C = [`c1`] at "`x.foo(d); // c3`", which has only one target `A:foo()`. Instead of passing `d` to `p` directly via one inter-procedural `assign` edge $d \xrightarrow[\hat{c3}]{\text{assign}}$ `p` as in L_{FC} (Eq. (3)), which is illustrated in Fig. 4, since L_{FC} requires `A:foo()` to be found separately outside L_{FC}, L_{DCR} passes `d` to `p` indirectly via a sequence of PAG edges, along the path given in Eq. (7) (shown also in Fig. 7), by discovering this dispatch target dynamically during the sub-path from `d` to `p`. Briefly, we address CHL1 by requiring `new[A]` to be matched with `dispatch[A]`. We address CHL2 by first storing `d` into a special field of `x` to trigger dynamic dispatch at this callsite and then loading it from the same special field of $\texttt{this}^{\texttt{A:foo()}}$ into `p` later (with the two sub-paths highlighted in). In addition, we perform dynamic dispatch correctly at this callsite under C = [`c1`] (with the sub-path highlighted in) as in the case of L_{FC}. We address CHL3 by passing `d` to `p` also correctly under [`c3`,`c1`], where `c3` records the callsite as in L_{FC} (Eq. (3)) for which parameter passing takes place and `c1` signifies further the context under which the receiver object `A1` flows into the receiver variable `x` at this callsite (with the sub-path highlighted in). The significance of the two below-edge labels, $\boxed{\hat{\texttt{c3}}}$ and $\boxed{\check{\texttt{c3}}}$, in addressing CHL3 cannot be over-emphasized. This ensures that if we start dynamic dispatch at callsite `c3` under context C = [`c1`], we will always return to the same callsite under the same context even though `c1` is lost (i.e., balanced out due to $\hat{\texttt{c1}}\check{\texttt{c1}}$) just after `x` $\xrightarrow[\check{\texttt{c1}}]{\overline{\text{assign}}}$ `a` at the beginning of the traversal for performing dynamic dispatch at callsite `c3`. To address CHL1 – CHL3, we have designed L_{DCR} to be the intersection of three CFLs operating on a new PAG representation as described below.

3 L_{DCR}: Design and Insights

When solving a CFL-reachability problem with a CFL, the CFL and its underlying graph structure are always inter-connected and carefully designed together. To break their cyclic

$$\frac{\texttt{x = }\textbf{new}\texttt{ T // O}}{O \xrightarrow{\mathsf{new[T]}} \texttt{x}}\ [\text{C-New}] \qquad \frac{\texttt{x = y}}{\texttt{y} \xrightarrow{\mathsf{assign}} \texttt{x}}\ [\text{C-Assign}]$$

$$\frac{\texttt{x = y.f}}{\texttt{y} \xrightarrow{\mathsf{load[f]}} \texttt{x}}\ [\text{C-Load}] \qquad \frac{\texttt{x.f = y}}{\texttt{y} \xrightarrow{\mathsf{store[f]}} \texttt{x}}\ [\text{C-Store}]$$

$$\frac{\texttt{x} = \texttt{m}(a_1, \ldots, a_n)\ \texttt{// c}}{\forall\ i \in [1, n] : a_i \xrightarrow[\hat{c}]{\mathsf{assign}} p_i^{\texttt{m}} \quad \texttt{ret}^{\texttt{m}} \xrightarrow[\check{c}]{\mathsf{assign}} \texttt{x}}\ [\text{C-SCall}]$$

$$\frac{\texttt{x} = \texttt{r.m}(a_1, \ldots, a_n)\ \texttt{// c} \quad \texttt{t} <: \texttt{DeclTypeOf(r)} \quad \texttt{m}' = \texttt{dispatch(m,t)}}{\begin{array}{c}\forall\ i \in [1, n] : a_i \xrightarrow[\boxed{\hat{c}}]{\mathsf{store}[i]} \texttt{r} \quad \texttt{r} \xrightarrow[\boxed{\check{c}}]{\mathsf{load[0]}} \texttt{x} \quad \texttt{r} \xrightarrow{\mathsf{assign}} \texttt{r\#c} \\ \texttt{r} \xrightarrow[\boxed{\check{c}}]{\mathsf{assign}} \texttt{r\#c} \quad \texttt{r\#c} \xrightarrow[\hat{c}]{\mathsf{dispatch[t]}} \texttt{this}^{\texttt{m}'}\end{array}}\ [\text{C-VCall}]$$

$$\frac{\texttt{M is an instance method} \quad p_i^{\texttt{M}} \text{ is its } i\text{th parameter}}{\texttt{this}^{\texttt{M}} \xrightarrow{\mathsf{load}[i]} p_i^{\texttt{M}}}\ [\text{C-Param}] \qquad \frac{\texttt{M is an instance method}}{\texttt{ret}^{\texttt{M}} \xrightarrow{\mathsf{store[0]}} \texttt{this}^{\texttt{M}}}\ [\text{C-Ret}]$$

Fig. 6. Rules for building the PAG required by L_{DCR}.

dependencies, we first describe how to represent a program with a new PAG representation to facilitate on-the-fly call graph construction (Sec. 3.1). We then formalize L_{DCR} by explaining how we address the three challenges (CHL1 – CHL3) and providing some insights in understanding its design (Sec. 3.2).

3.1 Pointer Assignment Graph

For a program, we use the rules given in Fig. 6 to build a PAG representation for L_{DCR}. We first introduce these rules briefly by highlighting their differences from those adopted by L_{FC} (Fig. 2) for building an L_{FC}-oriented PAG representation. We then illustrate these rules with an example. Note that one is expected to develop a reasonably good understanding of this new PAG representation before examining the three CFLs used for defining L_{DCR} later.

As in the case of L_{FC} (Fig. 2), the inverse of a PAG edge is not given explicitly. For each PAG edge $x \xrightarrow[c]{\ell} y$, its inverse edge is defined as $y \xrightarrow[\bar{c}]{\bar{\ell}} x$ as in L_{FC} exactly (Sec. 2.1.2), except that a below-edge label can also be $\boxed{\hat{c}}$ or $\boxed{\check{c}}$ (in addition to $\hat{c}$ and $\check{c}$), in which case, $\overline{\boxed{\hat{c}}} = \boxed{\check{c}}$ and $\overline{\boxed{\check{c}}} = \boxed{\hat{c}}$, where c identifies a callsite. To trigger dynamic dispatch at a callsite c, an edge with a boxed below-edge label also represents conceptually a new kind of inter-procedural value-flow entering into (marked by $\boxed{\hat{c}}$) or exiting from (marked by $\boxed{\check{c}}$) from a method invoked at c. Such boxed below-edge labels are introduced for addressing CHL3 only and their significance will become clear in Sec. 3.2.2.

Our PAG representation (for supporting L_{DCR}) differs from that for supporting L_{FC} (Fig. 2) mainly in how virtual callsites are handled. Therefore, [C-Assign], [C-Load], and [C-Store] are identical to [P-Assign], [P-Load], and [P-Store], respectively. In addition, [C-SCall], which behaves also identically as [P-SCall], handles parameter passing at a static callsite c simply as assignments in terms of inter-procedural `assign` edges, with its entry (exit) context being $\hat{c}$ ($\check{c}$).

[C-New], [C-VCall], [C-Param], and [C-Ret] build the PAG edges together to enable L_{DCR} to perform its own on-the-fly call graph construction at virtual callsites (for addressing CHL1 and CHL2). In [C-New] (unlike [P-New] in Fig. 2), $O \xrightarrow{\mathsf{new[T]}} \mathtt{x}$ encodes explicitly the dynamic type `T` of O in order to support dynamic dispatch on O while also enabling O to be passed as a receiver object to a method (dispatched on O) without the precision loss discussed using the code in Fig. 5.

Given an instance method `M` (with $\mathtt{this}^{\mathtt{M}}$ denoting its `this` variable), its i-th (non-`this`) parameter $p_i^{\mathtt{M}}$ (where i starts from 1) is modeled as a special field of $\mathtt{this}^{\mathtt{M}}$ (identified by offset i) and its return variable $\mathtt{ret}^{\mathtt{M}}$ also as a special field of $\mathtt{this}^{\mathtt{M}}$ (identified by offset 0). Thus, we can initialize $p_i^{\mathtt{M}}$ with a load $\mathtt{this}^{\mathtt{M}} \xrightarrow{\mathsf{load}[i]} p_i^{\mathtt{M}}$ ([C-Param]) and $\mathtt{this}^{\mathtt{M}}.0$ with a store $\mathtt{ret}^{\mathtt{M}} \xrightarrow{\mathsf{store[0]}} \mathtt{this}^{\mathtt{M}}$ ([C-Ret]).

[C-VCall], which is the most complex rule, differs fundamentally from [P-VCall] (Fig. 2) in handling a virtual callsite "$\mathtt{x} = \mathtt{r.m}(a_1, \ldots, a_n)$ // c". For convenience, we make use of `r#c` (a temporary) to identify uniquely this particular occurrence of `r` at this callsite. When building the PAG, we over-approximate the set of target methods invoked at each callsite (and consequently, the call graph for the program) by using class hierarchy analysis (CHA) [9] (due to $\mathtt{t} <: \mathtt{DeclTypeOf(r)}$ and $\mathtt{m}' = \mathtt{dispatch(m,t)}$). It is crucial to highlight that (1) CHA relies solely on the type information of a program and has the ability to construct an initial call graph in a linear manner; and (2) L_{DCR} will perform its on-the-fly call graph construction over such an over-approximated PAG with spurious call targets being filtered out (as discussed at the end of Sec. 3.2). To pass a non-receiver argument a_i ($1 \leqslant i \leqslant n$) to the corresponding parameter $p_i^{\mathtt{m}'}$ of a target method, `m′`, we make use of a store $a_i \xrightarrow[\hat{\boxed{c}}]{\mathsf{store}[i]} \mathtt{r}$ introduced in this rule and a matching load $\mathtt{this}^{\mathtt{m}'} \xrightarrow{\mathsf{load}[i]} p_i^{\mathtt{m}'}$ ([C-Param]). By performing CFL-reachability under L_{DCR}, traversing such an edge will trigger a search for the dynamic type of every receiver object pointed to by `r` (marked by $\hat{\boxed{c}}$). Encountering $\mathtt{r} \xrightarrow[\check{\boxed{c}}]{\mathsf{assign}} \mathtt{r\#c} \xrightarrow[\hat{c}]{\mathsf{dispatch[t]}} \mathtt{this}^{\mathtt{m}'}$ (introduced in this rule) later signifies that one such a dynamic type `t` has been found (marked by $\check{\boxed{c}}$) so that `m′`, where $\mathtt{m}' = \mathtt{dispatch(m,t)}$, can be dispatched with $\hat{c}$ as its entry context (as desired), where c is recovered from $\boxed{c}$. By definition, a `dispatch` edge also serves as an `assign` edge as well. As for the receiver variable `r`, we use $\mathtt{r} \xrightarrow{\mathsf{assign}} \mathtt{r\#}c$ (without a need for relating `r` to itself). Finally, we assign $\mathtt{ret}^{\mathtt{m}'}$ (saved earlier in $\mathtt{this}^{\mathtt{m}}.0$) ([C-Ret]) to `x` via a load $a_0 \xrightarrow[\check{\boxed{c}}]{\mathsf{load[0]}} \mathtt{x}$ (introduced in this rule), where $\check{\boxed{c}}$ signifies the end of dynamic dispatch on `r` on exit from callsite c.

Fig. 7 depicts the PAG used by L_{DCR} for our motivating example given in Fig. 3. As shown, this PAG (referred to below), which is designed to enable L_{DCR} to perform its own built-in on-the-fly call graph construction, is fundamentally different from the PAG (Fig. 4) used by L_{FC}.

3.2 L_{DCR}

We express L_{DCR} as the intersection of three CFLs, $L_{DCR} = L_D \cap L_C \cap L_R$, with each specifying a different aspect of kCFA. In Sec. 3.2.1, we introduce L_D, which describes field accesses

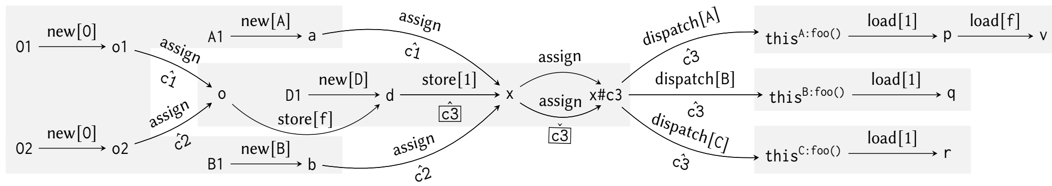


Fig. 7. The PAG for L_{DCR} constructed for the program given in Fig. 3.

and dynamic dispatch, for addressing CHL1 and CHL2 (Sec. 3.2.1). L_C, which is given in Eq. (2), enforces callsite-sensitivity by using the below-edge terminals $\hat{c}$ and $\check{c}$ in the standard manner. In Sec. 3.2.2, we introduce L_R (defined over the terminals $\hat{c}$ and $\check{c}$, $\boxed{\hat{c}}$, and $\boxed{\check{c}}$), to ensure that parameter passing happens correctly in the presence of on-the-fly call graph construction, for addressing CHL3. We shall restrict our discussions to parameter passing as method returns are handled in the same way. When introducing the grammars for L_D and L_R below, all the double-label PAG edges in a given PAG are handled in the same way as we have done to L_F and L_C, as described in Sec. 2.1.2.

We shall speak of an L_{DC}-path as we do for L_{FC}-path (Sec. 2.2.3). Similarly, we shall also speak of an L_{DCR}-path p with the understanding that $L_D(p) \in L_D$, $L_C(p) \in L_C$, and $L_R(p) \in L_R$.

To incorporate call graph construction into a CFL-reachability formulation, as defined below, it is necessary to ensure its soundness by guaranteeing that it provides an over-approximation of the set of target methods resolved at a callsite. This means that the constructed call graph should include all possible target methods that could be called at a given callsite. Furthermore, precision is maintained by excluding any spurious target methods from the resolution, ensuring that only relevant and accurate target methods are considered.

Definition 1 (Soundness and Precision). Let L be a CFL-reachability formulation that differs from L_{FC} only in how they handle parameter passing at virtual callsites, where L itself specifies dynamic method dispatch at virtual callsites (i.e., performs its own built-in call graph construction) and L_{FC} relies on a separate algorithm $\mathcal{A}$ (by, e.g., calling L_{FC} recursively to find the receiver objects of virtual callsites under some given contexts) for performing call graph construction on the fly (Sec. 2.2.3.2). Consider a virtual callsite "$\mathtt{r.m}(a_1, \ldots, a_n)$; // c", where parameter passing takes place under a given context C. Let T be the set of target methods found by $\mathcal{A}$ at this callsite under C (i.e., found also by L_{FC} if a points-to query is issued for r for this callsite under C separately) so that L_{FC} can perform parameter passing for these methods. Then L is sound if it enables parameter passing to be performed under C for at least all the target methods in T and L is precise (in addition to being sound) if it enables parameter passing under C for exactly the same target methods in T.

We shall see that L_{DC} is sound (but imprecise) but L_{DCR} is precise (and obviously sound).

Let us consider parameter passing for an argument at a virtual call site "$\mathtt{r.m}(a_1, \ldots, a_n)$; //c" under a given context C. In the special case when the argument is the receiver variable r, which points directly to a set of receiver objects, we only need to address CHL1 by passing a receiver object to a target method m' when m' can be dispatched on the receiver object. If the argument is a_i, the basic idea in addressing CHL2 and CHL3 (facilitated by the PAG designed according to the rules given in Fig. 6) is to first store a_i into $\mathtt{r}.i$ (at its special field i), then discover the dynamic type t of every receiver object pointed to by r under C and propagate `t` to the callsite `c` where $\mathtt{m}' = \mathtt{dispatch}(\mathtt{m}, \mathtt{t})$, and finally, assign $\mathtt{this}^{\mathtt{m}'}.i$ to $p_i^{\mathtt{m}'}$ for this callsite under still the same given context C. As discussed earlier, method returns are handled similarly.

Let us revisit our motivating example with its PAG depicted in Fig. 7. With L_{DCR}, we will ensure that the PAG contains a unique L_{DCR}-path from `O1` to `v`, as depicted in Eq. (7), so that `v` can point to `O1` only when `bar()` is called at callsite `c1`. The sub-path from `O1` to `d` indicates that `O1` is stored into `d.f`, where `d` points to `D1`, due to the call "`bar(a, o1); // c1`". The sub-path from `d` to `p` reveals parameter passing from `d` to `p` at "`x.foo(d); // c3`" for the target method `A:foo()` discovered on the fly by L_{DCR} itself under C = [`c1`]. In Sec. 2.2.4, we have already discussed how we address CHL1 – CHL3 at this callsite. We wish to add now that $\hat{\boxed{\mathtt{c3}}}$ and $\check{\boxed{\mathtt{c3}}}$ mark the beginning and end of dynamic dispatch performed at this callsite for `d`, respectively. During the CFL-reachability traversal between $\hat{\boxed{\mathtt{c3}}}$ and $\check{\boxed{\mathtt{c3}}}$, we discover that `x` points to `A1` of type `A` under [`c1`] must return to `x` under also [`c1`]. Given the receiver object `A1` found, we can then dispatch `A:foo()` via $\mathtt{x\#c3} \xrightarrow[\hat{\mathtt{c3}}]{\mathtt{dispatch[A]}} \mathtt{this}^{\mathtt{A:foo()}}$ so that `d` can be passed to `p` under [`c3`, `c1`], with `c3` recovered from $\boxed{\mathtt{c3}}$. While L_{FC} [54] uses [`c3`] for passing `d` to `p` (Eq. (3)), L_{DCR} uses [`c3`, `c1`] more specifically to indicate that this happens only when `x` points to `A1` under [`c1`].

3.2.1 The L_D Language This CFL describes not only field-sensitive accesses as balanced parentheses as in L_F given in Eq. (1) but also dynamic dispatch in the language itself:

$$
\begin{aligned}
\mathsf{flowsto} &\longrightarrow \mathsf{new[t]}\ (\mathsf{flows} \mid \mathsf{dispatch[t]})^* \\
\mathsf{flows} &\longrightarrow \mathsf{assign} \mid \mathsf{store[f]}\ \mathsf{alias}\ \mathsf{load[f]} \\
\mathsf{alias} &\longrightarrow \overline{\mathsf{flowsto}}\ \mathsf{flowsto} \\
\overline{\mathsf{flowsto}} &\longrightarrow (\overline{\mathsf{dispatch[t]}} \mid \overline{\mathsf{flows}})^*\ \overline{\mathsf{new[t]}} \\
\overline{\mathsf{flows}} &\longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathsf{alias}\ \overline{\mathsf{store[f]}}
\end{aligned}
\tag{8}
$$

All the below-edge labels are handled similarly as in the case of L_F. By decomposing each double-label edge into two single-label edges as described in Sec. 2.1.2, flows becomes $\mathsf{flows} \longrightarrow ... \mid \hat{c} \mid \check{c} \mid \hat{\boxed{c}} \mid \check{\boxed{c}}$ and $\overline{\mathsf{flows}}$ becomes $\overline{\mathsf{flows}} \longrightarrow ... \mid \hat{c} \mid \check{c} \mid \hat{\boxed{c}} \mid \check{\boxed{c}}$ in their full generality.

In designing L_D, we have extended L_F [54, 55] by preserving its capability in handling field accesses as balanced parentheses and adding a new capability in supporting dynamic dispatch, and consequently, on-the-fly call graph construction. Below we describe separately how L_D is designed to address CHL1 and CHL2 and why we have selected such a dynamic dispatch approach.

3.2.1.1 CHL1 In addressing this first challenge concerning parameter passing at a virtual callsite, we must distinguish its receiver variable from its other arguments to ensure that the receiver objects pointed by the receiver variable can only be passed to the `this` variable of a method that can be dispatched on these receiver objects. Consider `x.foo(`**`null`**`)` in the code snippet given in Fig. 5, where `x` may point to both `A1` and `B1`. The traditional L_{FC}-based formulation that uses a separate algorithm for call graph construction (Fig. 2) will end up passing both `A1` and `B1` to $\mathtt{this}^{\mathtt{A:foo()}}$ due to the existence of the two L_{FC}-paths given in Eq. (5) and Eq. (6), although `B1` is spurious (Fig. 1).

In L_D, we make explicit the dynamic types of objects in the four kinds of terminals, $\mathsf{new[t]}$, $\overline{\mathsf{new[t]}}$, $\mathsf{dispatch[t]}$, and $\overline{\mathsf{dispatch[t]}}$. During a $\mathsf{flowsto}$ ($\overline{\mathsf{flowsto}}$) traversal, we require the type information in $\mathsf{dispatch[t]}$ ($\overline{\mathsf{dispatch[t]}}$) to be consistent with that in its corresponding $\mathsf{new[t]}$ ($\overline{\mathsf{new[t]}}$). As a result, the two L_{FC}-paths in Eq. (5) and Eq. (6) become:

$$\mathsf{A1} \xrightarrow{\mathsf{new[A]}} \mathsf{a} \xrightarrow{\mathsf{store[f]}} \mathsf{d} \xrightarrow{\overline{\mathsf{new[D]}}} \mathsf{D1} \xrightarrow{\mathsf{new[D]}} \mathsf{d} \xrightarrow{\mathsf{load[f]}} \mathsf{x} \xrightarrow{\mathsf{assign}} \mathsf{x\#c} \xrightarrow[\hat{\mathsf{c}}]{\mathsf{dispatch[A]}} \mathtt{this}^{\mathsf{A:foo()}}$$

$$\mathsf{B1} \xrightarrow{\mathsf{new[B]}} \mathsf{b} \xrightarrow{\mathsf{store[f]}} \mathsf{d} \xrightarrow{\overline{\mathsf{new[D]}}} \mathsf{D1} \xrightarrow{\mathsf{new[D]}} \mathsf{d} \xrightarrow{\mathsf{load[f]}} \mathsf{x} \xrightarrow{\mathsf{assign}} \mathsf{x\#c} \xrightarrow[\hat{\mathsf{c}}]{\mathsf{dispatch[A]}} \mathtt{this}^{\mathsf{A:foo()}}$$

The first is an L_D-path since $\mathsf{new[A]}\ \mathsf{flows}^*\ \mathsf{dispatch[A]} \in L_D$, but the second is not an L_D-path since $\mathsf{new[B]}\ \mathsf{flows}^*\ \mathsf{dispatch[A]} \notin L_D$. Thus, B1 cannot flow to $\mathtt{this}^{\mathsf{A:foo()}}$ spuriously.

In Eq. (7), A1 can be passed to $\mathtt{this}^{\mathsf{A:foo()}}$ since `A:foo()` can be dispatched on A1.

Lemma 1. Consider a virtual callsite $\mathtt{x} = \mathtt{r.m}(a_1, \ldots, a_n)$ in Java. In L_D, every receiver object pointed to by `r` flows only to the `this` variable of a method that can be dispatched on the receiver object.

Proof Sketch. Follows from the definition of L_D. □

3.2.1.2 CHL2 In addressing this second challenge, we must decide how to trigger dynamic dispatch during the course of parameter passing at a virtual callsite. We accomplish this by using $L_{DC} = L_D \cap L_C$. To understand our approach, we examine again the L_{DCR}-path given in Eq. (7). For convenience, we duplicate it below by ignoring $\boxed{\hat{\mathsf{c3}}}$ and $\boxed{\check{\mathsf{c3}}}$ to obtain the following L_{DC}-path:

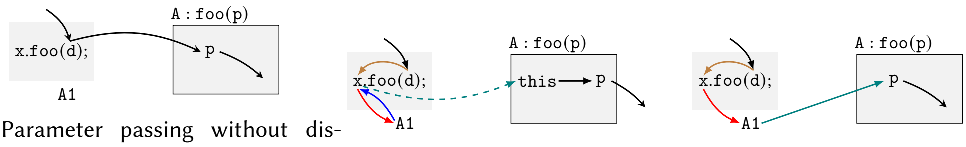
$$\begin{array}{l}\mathsf{O1} \xrightarrow{\mathsf{new[O]}} \mathsf{o1} \xrightarrow[\hat{\mathsf{c1}}]{\mathsf{assign}} \mathsf{o} \xrightarrow{\mathsf{store[f]}} \mathsf{d} \xrightarrow{\overline{\mathsf{new[D]}}} \mathsf{D1} \xrightarrow{\mathsf{new[D]}} \mathsf{d} \xrightarrow{\mathsf{store[1]}} \mathsf{x} \xrightarrow[\check{\mathsf{c1}}]{\overline{\mathsf{assign}}} \mathsf{a} \xrightarrow{\overline{\mathsf{new[A]}}} \mathsf{A1} \\ \quad \xrightarrow{\mathsf{new[A]}} \mathsf{a} \xrightarrow[\hat{\mathsf{c1}}]{\mathsf{assign}} \mathsf{x} \xrightarrow{\mathsf{assign}} \mathsf{x\#c3} \xrightarrow[\hat{\mathsf{c3}}]{\mathsf{dispatch[A]}} \mathtt{this}^{\mathsf{A:foo()}} \xrightarrow{\mathsf{load[1]}} \mathsf{p} \xrightarrow{\mathsf{load[f]}} \mathsf{v}\end{array} \tag{9}$$

To find whether O1 can flow into `v` starting from "`bar(a,o1); // c1`", we need to perform parameter passing for `d` at "`x.foo(d); // c3`" under context C = [c1]. This is achieved by traversing the sub-path from `d` to the parameter `p` of `A:foo()`. We start with a store $\mathsf{d} \xrightarrow{\mathsf{store[1]}} \mathsf{x}$ to trigger a $\overline{\mathsf{flowsto}}$ traversal via $\mathsf{x} \xrightarrow[\check{\mathsf{c1}}]{\overline{\mathsf{assign}}} \mathsf{a} \xrightarrow{\overline{\mathsf{new[A]}}} \mathsf{A1}$ under the given context C = [c1], return to `x` backwards via $\mathsf{A1} \xrightarrow{\mathsf{new[A]}} \mathsf{a} \xrightarrow[\hat{\mathsf{c1}}]{\mathsf{assign}} \mathsf{x}$, dispatch at the callsite via $\mathsf{x} \xrightarrow{\mathsf{assign}} \mathsf{x\#c3} \xrightarrow[\hat{\mathsf{c3}}]{\mathsf{dispatch[A]}} \mathtt{this}^{\mathsf{A:foo()}}$, and finally, pass `d` to `p` via a load $\mathtt{this}^{\mathsf{A:foo()}} \xrightarrow{\mathsf{load[1]}} \mathsf{p}$. While L_{FC} passes `d` to `p` in Eq. (3) by one assign edge $d \xrightarrow[\hat{\mathsf{c3}}]{\mathsf{assign}} \mathsf{p}$ under [c3], L_{DC} passes `d` to `p` via a sequence of PAG edges under [c3,c1] (indicating that this parameter passing happens only when `x` point to A1 under [c1]).

According to Definition 1, L_{DC} is sound if it can perform parameter passing for a superset of the set of target methods that can be possibly dispatched at a virtual callsite.

Lemma 2. L_{DC} is sound in handling parameter passing at every virtual callsite.

Proof Sketch. Let "$\mathtt{r.m}(a_1, \ldots, a_n)$; // c" be a fixed but arbitrary virtual callsite, where parameter passing for one of its arguments takes place under a given context C. Let T be the set of target methods found on the fly at this callsite under C by a separate call



(a) Parameter passing without dispatch (b) Dispatch at callsite (c) Dispatch at allocation site

Fig. 8. Three approaches for performing dynamic dispatch at a virtual callsite during parameter passing.

graph construction algorithm used in L_{FC}. As $\mathtt{r}$ is handled similarly as in L_{FC}, it suffices to consider parameter passing for a non-receiver-variable argument a_i. Due to the existence of $a_i \xrightarrow{\mathsf{store}[i]} \mathtt{r}$, L_{DC} will perform dynamic dispatch by finding the receiver objects pointed to by $\mathtt{r}$ under also C. As L_{DC} differs from L_{FC} only in their handling of parameter passing at virtual callsites, the set of target methods found by L_{DC} must include T. In addition, for every target $m' \in T$, there always exists a path q in the PAG:

$$a_i \xrightarrow{\mathsf{store}[i]} \mathtt{r}\ \overline{\mathsf{flowsto}}\ O\ \mathsf{flowsto}\ \mathtt{r} \xrightarrow{\mathsf{assign}} \mathtt{r\#c} \xrightarrow[\hat{c}]{\mathsf{dispatch}[\_]} \mathtt{this}^{m'} \xrightarrow{\mathsf{load}[i]} p_i \qquad (10)$$

where p_i is the i-th parameter of m'. Here, if we write u to represent the $\overline{\mathsf{flowsto}}$-path "$\mathtt{r}\ \overline{\mathsf{flowsto}}\ O$", then the flows-path "$O\ \mathsf{flowsto}\ \mathtt{r}$" is its inverse $\overline{u}$. By construction, a_i flows p_i according to L_D and $L_C(q) \in L_C$ according to L_C. In addition, $L_C(q)$ is guaranteed to be a sequence of (calling) contexts that can happen under C since u is traversed under C. Therefore, L_{DC} is sound by Definition 1. □

Below we discuss three possible dynamic dispatch approaches, illustrated in Fig. 8, for handling parameter passing and explain why L_D is designed to adopt a callsite-based approach (Fig. 8b).

As discussed in Sec. 2.2.3, L_{FC} [54] solves kCFA by using a separate algorithm for call graph construction (Fig. 8a) and may thus cause kCFA to lose precision (either directly (Sec. 2.2.3.1 and Sec. 2.2.3.2) or resorting to a pre-analysis [36] (discussed in Sec. 2.2.3.3 and evaluated in Sec. 4.2)).

In L_D, passing an argument `d` at `x.foo(d)` to a parameter will trigger immediately a $\overline{\mathsf{flowsto}}$ traversal looking for a receiver object of `x`, as symbolized by a red arrow ($\longrightarrow$) in Fig. 8b (for performing dynamic dispatch at this callsite) and Fig. 8c (for performing dynamic dispatch at the allocation site of the receiver object). L_D adopts the former approach since the latter is infeasible.

Let us explain why the allocation-site-based dispatch (Fig. 8c) is infeasible. To handle parameter passing only (without considering method returns here), we need to extend L_F to become:

$$\begin{array}{lcl} \mathsf{flows} & \longrightarrow & \cdots \mid \underline{\mathsf{store}[\mathtt{m:i}]}\ \overline{\mathsf{flowsto}}\ \underline{\mathsf{load}[\mathtt{m:i}]} \\ \mathsf{flows} & \longrightarrow & \cdots \mid \overline{\mathsf{load}[\mathtt{m:i}]}\ \mathsf{flowsto}\ \overline{\mathsf{store}[\mathtt{m:i}]} \end{array} \qquad (11)$$

where $\mathsf{store}[\mathtt{m:i}]$ ($\mathsf{load}[\mathtt{m:i}]$) is used to replace $\mathsf{store}[\mathtt{i}]$ ($\mathsf{load}[\mathtt{i}]$) in Fig. 6 in order to encode also the signature of a method invoked at $\mathtt{r.m}(a_1, \ldots, a_n)$. Thus, the L_{FC}-path in Eq. (3) becomes:

$$\mathsf{O1} \xrightarrow{\mathsf{new}} \mathsf{o1} \xrightarrow[\hat{c1}]{\mathsf{assign}} \mathsf{o} \xrightarrow{\mathsf{store}[\mathsf{f}]} \mathsf{d} \xrightarrow{\overline{\mathsf{new}}} \mathsf{D1} \xrightarrow{\mathsf{new}} \mathsf{d} \xrightarrow{\mathsf{store}[\mathsf{foo:1}]} \mathsf{x} \xrightarrow[\check{c1}]{\overline{\mathsf{assign}}} \mathsf{a} \xrightarrow{\overline{\mathsf{new}}} \mathsf{A1} \xrightarrow{\mathsf{load}[\mathsf{foo:1}]} \mathsf{p} \xrightarrow{\mathsf{load}[\mathsf{f}]} \mathsf{v} \qquad (12)$$

```
static void main() {
  A a = new A(); // A1
  O o1 = new O(); // O1
  O o2 = new O(); // O2
  a.m(o1); // c1
  a.n(o2); } // c2
class O { }
class A {
  void m(Object p) { ... }
  void n(Object q) { ... } }
```

Fig. 9. An example for illustrating the imprecision of L_{DC} caused by an incorrect dispatch site.

```
class A {
  O id(O p) { return p; } }
class O { }
static O wid(A a, O o) {
  O v = a.id(o); // c3
  return v;
}
static void main() {
  A a1 = new A(); // A1
  O o1 = new O(); // O1
  O o2 = new O(); // O2
  O v1 = wid(a1, o1); // c1
  O v2 = wid(a1, o2); // c2
}
```

Fig. 10. An example for illustrating the imprecision of L_{DC} caused by an incorrect dispatch context.

where we have $\mathtt{d} \xrightarrow{\mathtt{store[foo:1]}} \mathtt{x}$ ($\rightarrow$), $\mathtt{x} \xrightarrow[\check{\mathtt{c1}}]{\overline{\mathtt{assign}}} \mathtt{a} \xrightarrow{\overline{\mathtt{new}}} \mathtt{A1}$ ($\rightarrow$), and $\mathtt{A1} \xrightarrow{\mathtt{load[foo:1]}} \mathtt{p}$ ($\rightarrow$). However, O1 flows to v under [] incorrectly (with $\hat{\mathtt{c1}}$ and $\check{\mathtt{c1}}$ being matched). Due to the nature of object-sensitivity (where receiver objects are used as context elements), the CFL-reachability formulation for object-sensitive pointer analysis [35, 37] (as reviewed briefly in Sec. 5.1) works well with the allocation-site-based dispatch approach.

3.2.2 The L_R Language L_{DC} is sound but imprecise. We use two examples given in Fig. 9 and Fig. 10 to illustrate the imprecision of L_{DC} and highlight the two roles that L_R plays in L_{DCR}.

L_{DC} can lose precision caused by an incorrect dispatch callsite. Consider the following two L_{DC}-paths in the PAG of Fig. 9 (by ignoring the boxed labels $\boxed{\hat{\mathtt{c1}}}$, $\boxed{\check{\mathtt{c1}}}$ and $\boxed{\check{\mathtt{c2}}}$ for now):

$$\mathtt{O1} \xrightarrow{\mathtt{new[0]}} \mathtt{o1} \xrightarrow[\boxed{\hat{\mathtt{c1}}}]{\mathtt{store[1]}} \mathtt{a} \xrightarrow{\overline{\mathtt{new[A]}}} \mathtt{A1} \xrightarrow{\mathtt{new[A]}} \mathtt{a} \xrightarrow[\boxed{\check{\mathtt{c1}}}]{\mathtt{assign}} \mathtt{a\#c1} \xrightarrow[\hat{\mathtt{c1}}]{\mathtt{dispatch[A]}} \mathtt{this}^{\mathtt{m}} \xrightarrow{\mathtt{load[1]}} \mathtt{p} \tag{13}$$

$$\mathtt{O1} \xrightarrow{\mathtt{new[0]}} \mathtt{o1} \xrightarrow[\boxed{\hat{\mathtt{c1}}}]{\mathtt{store[1]}} \mathtt{a} \xrightarrow{\overline{\mathtt{new[A]}}} \mathtt{A1} \xrightarrow{\mathtt{new[A]}} \mathtt{a} \xrightarrow[\boxed{\check{\mathtt{c2}}}]{\mathtt{assign}} \mathtt{a\#c2} \xrightarrow[\hat{\mathtt{c2}}]{\mathtt{dispatch[A]}} \mathtt{this}^{\mathtt{n}} \xrightarrow{\mathtt{load[1]}} \mathtt{q} \tag{14}$$

These two L_{DC}-paths both keep track of where O1 flows to in the PAG of this program. According to the first L_{DC}-path, O1 flows to p as expected. However, due to the existence of the second L_{DC}-path, O1 can also flow to q spuriously as the $\overline{\mathtt{flowsto}}$ traversal for finding the receiver object of a is triggered at callsite c1 but the dispatch ends up happening at callsite c2. To eliminate such precision loss, L_R requires boxed edge labels to be matched as balanced parentheses. As a result, the first L_{DC}-path in Eq. (13) will be considered as a valid L_{DCR}-path (since $\boxed{\hat{\mathtt{c1}}}$ is matched by $\boxed{\check{\mathtt{c1}}}$) but the second L_{DC}-path in Eq. (14) will be ruled out (since $\boxed{\hat{\mathtt{c1}}}$ is not matched by $\boxed{\check{\mathtt{c2}}}$).

L_{DC} can also lose precision caused by an incorrect dispatch context. Consider the following two L_{DC}-paths in the PAG of Fig. 10 (by ignoring the boxed labels $\boxed{\hat{\mathtt{c3}}}$ and $\boxed{\check{\mathtt{c3}}}$ for now):

$$
\begin{aligned}
&\mathtt{O1} \xrightarrow{\mathsf{new[0]}} \mathtt{o1} \xrightarrow[\hat{\mathtt{c1}}]{\mathsf{assign}} \mathtt{o} \xrightarrow[\boxed{\hat{\mathtt{c3}}}]{\mathsf{store[1]}} \mathtt{a} \xrightarrow[\check{\mathtt{c1}}]{\overline{\mathsf{assign}}} \mathtt{a1} \xrightarrow{\overline{\mathsf{new[A]}}} \mathtt{A1} \xrightarrow{\mathsf{new[A]}} \mathtt{a1} \xrightarrow[\hat{\mathtt{c1}}]{\mathsf{assign}} \mathtt{a} \\
&\xrightarrow[\boxed{\check{\mathtt{c3}}}]{\mathsf{assign}} \mathtt{a\#c3} \xrightarrow[\hat{\mathtt{c3}}]{\mathsf{dispatch[A]}} \mathtt{this}^{\mathtt{id}} \xrightarrow{\mathsf{load[1]}} \mathtt{p} \xrightarrow{\mathsf{store[0]}} \mathtt{this}^{\mathtt{id}} \xrightarrow[\check{\mathtt{c3}}]{\overline{\mathsf{dispatch[A]}}} \mathtt{a\#c3} \\
&\xrightarrow[\boxed{\hat{\mathtt{c3}}}]{\overline{\mathsf{assign}}} \mathtt{a} \xrightarrow[\check{\mathtt{c1}}]{\overline{\mathsf{assign}}} \mathtt{a1} \xrightarrow{\overline{\mathsf{new[A]}}} \mathtt{A1} \xrightarrow{\mathsf{new[A]}} \mathtt{a1} \xrightarrow[\hat{\mathtt{c1}}]{\mathsf{assign}} \mathtt{a} \xrightarrow[\boxed{\check{\mathtt{c3}}}]{\mathsf{load[0]}} \mathtt{v} \xrightarrow[\check{\mathtt{c1}}]{\mathsf{assign}} \mathtt{v1}
\end{aligned}
\tag{15}
$$

$$
\begin{aligned}
&\mathtt{O1} \xrightarrow{\mathsf{new[0]}} \mathtt{o1} \xrightarrow[\hat{\mathtt{c1}}]{\mathsf{assign}} \mathtt{o} \xrightarrow[\boxed{\hat{\mathtt{c3}}}]{\mathsf{store[1]}} \mathtt{a} \xrightarrow[\check{\mathtt{c1}}]{\overline{\mathsf{assign}}} \mathtt{a1} \xrightarrow{\overline{\mathsf{new[A]}}} \mathtt{A1} \xrightarrow{\mathsf{new[A]}} \mathtt{a1} \xrightarrow[\hat{\mathtt{c2}}]{\mathsf{assign}} \mathtt{a} \\
&\xrightarrow[\boxed{\check{\mathtt{c3}}}]{\mathsf{assign}} \mathtt{a\#c3} \xrightarrow[\hat{\mathtt{c3}}]{\mathsf{dispatch[A]}} \mathtt{this}^{\mathtt{id}} \xrightarrow{\mathsf{load[1]}} \mathtt{p} \xrightarrow{\mathsf{store[0]}} \mathtt{this}^{\mathtt{id}} \xrightarrow[\check{\mathtt{c3}}]{\overline{\mathsf{dispatch[A]}}} \mathtt{a\#c3} \\
&\xrightarrow[\boxed{\hat{\mathtt{c3}}}]{\overline{\mathsf{assign}}} \mathtt{a} \xrightarrow[\check{\mathtt{c2}}]{\overline{\mathsf{assign}}} \mathtt{a1} \xrightarrow{\overline{\mathsf{new[A]}}} \mathtt{A1} \xrightarrow{\mathsf{new[A]}} \mathtt{a1} \xrightarrow[\hat{\mathtt{c2}}]{\mathsf{assign}} \mathtt{a} \xrightarrow[\boxed{\check{\mathtt{c3}}}]{\mathsf{load[0]}} \mathtt{v} \xrightarrow[\check{\mathtt{c2}}]{\mathsf{assign}} \mathtt{v2}
\end{aligned}
\tag{16}
$$

These two L_{DC}-paths differ only in their underlying contexts and target variables used: the second can be obtained from the first by replacing each occurrence of `c1` with `c2` and `v1` with `v2`. Both L_{DC}-paths keep track of where `O1` will flow to, starting from the call "`wid(a1,o1); // c1`". According to the first L_{DC}-path, `v1` points to `O1` as expected. However, due to the existence of the second L_{DC}-path, `O1`, which comes from callsite `c1`, will flow into `v2` at callsite `c2` spuriously. Consider the dynamic dispatch that happens at "`a.id(o); // c3`" due to the call "`wid(a1,o1); // c1`". In the first L_{DC}-path, `a` starts with pointing to `A1` under [`c1`] during its $\overline{\mathsf{flowsto}}$ traversal (to find what `a` points to) and ends up with pointing to `A1` under [`c1`] during the ensuing flowsto traversal. This flowsto traversal can happen from the call "`wid(a1,o1); // c1`". However, in the second L_{DC}-path, `a` starts also with pointing to `A1` under [`c1`] during its $\overline{\mathsf{flowsto}}$ traversal but ends up with pointing to `A1` under [`c2`] during the ensuing flowsto traversal. This flowsto traversal cannot happen from the call "`wid(a1,o1); // c1`".

Consider a virtual callsite "$\mathtt{r.m}(a_1, \ldots, a_n)$; `//c`" with a reference to Eq. (10). In general, when performing a $\overline{\mathsf{flowsto}}$ traversal to find that `r` points to a receiver object O under $[\check{c_1}, ..., \check{c_k}]$, L_R must be designed to ensure that we can return from O to `r` by performing a flowsto traversal under exactly $[\hat{c_k}, ..., \hat{c_1}]$ in order to avoid passing arguments spuriously.

To address CHL3 (i.e., the two sources of imprecision above), we introduce a third CFL L_R:

$$
\begin{aligned}
\mathsf{recoveredCtx} &\longrightarrow \mathsf{recoveredCtx}\ \hat{c} \mid \mathsf{recoveredCtx}\ \check{c} \mid \mathsf{recoveredCtx}\ \mathsf{siteRecovered} \mid \epsilon \\
\mathsf{siteRecovered} &\longrightarrow \boxed{\hat{c}}\ \mathsf{ctxRecovered}\ \boxed{\check{c}} \\
\mathsf{ctxRecovered} &\longrightarrow \mathsf{matched}\ \mathsf{ctxRecovered} \mid \mathsf{ctxRecovered}\ \mathsf{matched} \mid \check{c}\ \mathsf{ctxRecovered}\ \hat{c} \mid \epsilon \\
\mathsf{matched} &\longrightarrow \mathsf{matched}\ \mathsf{matched} \mid \hat{c}\ \mathsf{matched}\ \check{c} \mid \mathsf{siteRecovered} \mid \epsilon
\end{aligned}
\tag{17}
$$

All the above-edge labels are handled similarly as in the case of L_C. By decomposing each double-label edge into a sequence of two single-label edges as described in Sec. 2.1.2, matched is extended by adding a new production matched $\longrightarrow$ above-edge-label, where the new non-terminal above-edge-label is defined in terms of all the original above-edge labels in a given PAG.

With L_R being incorporated into L_{DC}, the resulting language $L_{DCR} = L_D \cap L_C \cap L_R$ will be precise in handing parameter passing for virtual callsites. Let us return to the two paths given in Eq. (15) and Eq. (16) with $\boxed{\hat{\mathtt{c3}}}$ and $\boxed{\check{\mathtt{c3}}}$) being considered. The first one is an L_{DCR}-path but the second is not. The first one is an L_{DCR}-path, since we start the dynamic dispatch process at callsite c3 (marked by the first $\boxed{\hat{\mathtt{c3}}}$) under context [c1] and return to the same callsite under the same context at the end of this dynamic dispatch process (marked by the first $\boxed{\check{\mathtt{c3}}}$) even though c1 is lost, i.e., balanced out due to $\hat{\mathtt{c1}}\check{\mathtt{c1}}$ just after the first $\mathtt{a} \xrightarrow[\check{\mathtt{c1}}]{\mathsf{assign}} \mathtt{a1}$ edge is traversed. The second one is not an L_{DCR}-path, since even though we also start the dynamic dispatch process at callsite c3 (marked by the first $\boxed{\hat{\mathtt{c3}}}$) under context [c1], we end up returning to the same callsite under a different and thus incorrect context, [c2], at the end of this dynamic dispatch process (marked by the first $\boxed{\check{\mathtt{c3}}}$). As a result, L_{DCR} will conclude that O1 is pointed to by v1 but not by v2 as desired.

Below, we give a formal development of L_R and then prove the precision of L_{DCR}.

We can obtain the points-to set of a variable v, $\mathsf{PTS}(v, c_v)$, from L_{DC} as follows. Given an L_C-path p with its label being $L_C(p) = \ell_1, \ldots, \ell_n$, where each ℓ_i is an entry or exit context label in an inter-procedural assign edge, the inverse of p, i.e., $\overline{p}$ has the label $L_C(\overline{p}) = \overline{\ell_n}, \ldots, \overline{\ell_1}$. By splitting p into a sub-path p^{ex} followed by a sub-path p^{en}, we can define ${L_C}^{\mathsf{ex}}(p) = L_C(p^{\mathsf{ex}})$ and ${L_C}^{\mathsf{en}}(p) = L_C(p^{\mathsf{en}})$, where $L_C(p) = {L_C}^{\mathsf{ex}}(p){L_C}^{\mathsf{en}}(p)$, such that ${L_C}^{\mathsf{ex}}(p)$ (${L_C}^{\mathsf{en}}(p)$) is derived from exit (entry) in L_C's grammar (Eq. (2)). Let $s \in L_C$. Let $\mathscr{B}(s)$ return the canonical form of s with all its balanced contexts (i.e., parentheses) removed. If c is a string of exit contexts of the form $\check{c_1} \ldots \check{c_n}$, we write $\mathscr{E}(c) = [c_1, \ldots, c_n]$ to turn it into a context representation (by noting that $\mathscr{E}(\epsilon) = [\,]$).

Given an L_{DC}-path p starting from an object O to a variable v, we can deduce the following points-to relation with the contexts of O and v being spelt out clearly:

$$\langle O, \mathscr{E}(\mathscr{B}({L_C}^{\mathsf{ex}}(p)))\rangle \quad \in \quad \mathsf{PTS}(v, \mathscr{E}(\overline{\mathscr{B}({L_C}^{\mathsf{en}}(p))})) \tag{18}$$

Let us consider an example. Let $p_{\mathtt{O1,v}}$ be the L_{DC}-path in Eq. (7) (by ignoring $\boxed{\hat{\mathtt{c3}}}$ and $\boxed{\check{\mathtt{c3}}}$). By definition, $L_C(p_{\mathtt{O1,v}}) = \hat{\mathtt{c1}}\check{\mathtt{c1}}\hat{\mathtt{c1}}\hat{\mathtt{c3}}$, where $p^{\mathsf{ex}}_{\mathtt{O1,v}}$ can be interpreted as the sub-path from O1 to A1 and $p^{\mathsf{en}}_{\mathtt{O1,v}}$ as the sub-path from A1 to v. Thus, ${L_C}^{\mathsf{ex}}(p_{\mathtt{O1,v}}) = \hat{\mathtt{c1}}\check{\mathtt{c1}}$ and ${L_C}^{\mathsf{en}}(p_{\mathtt{O1,v}}) = \hat{\mathtt{c1}}\hat{\mathtt{c3}}$. Since $\mathscr{E}(\mathscr{B}(\hat{\mathtt{c1}}\check{\mathtt{c1}})) = \mathscr{E}(\epsilon) = [\,]$ and $\mathscr{E}(\overline{\mathscr{B}(\hat{\mathtt{c1}}\hat{\mathtt{c3}})}) = \mathscr{E}(\overline{\hat{\mathtt{c1}}\hat{\mathtt{c3}}}) = \mathscr{E}(\check{\mathtt{c3}}\check{\mathtt{c1}}) = [\mathtt{c3}, \mathtt{c1}]$, we have:

$$\langle \mathtt{O1}, [\,]\rangle \quad \in \quad \mathsf{PTS}(\mathtt{v}, [\mathtt{c3}, \mathtt{c1}])$$

L_{DC} can lose precision since, for some L_{DC}-paths, its sub-paths responsible for performing dynamic dispatch can be spurious. Consider a virtual callsite $\mathtt{r.m}(a_1, \ldots, a_n)$ // c. Before passing an argument a_i into (or receiving a return value from) a method invoked, L_{DC} performs dynamic dispatch by carrying out the following alias-related traversal on its receiver variable $\mathtt{r}$:

$$\cdots \xrightarrow[\boxed{\hat{c}}]{\ell} \mathtt{r}\ \overline{\mathsf{flowsto}}\ O\ \mathsf{flowsto}\ \mathtt{r}' \xrightarrow[\boxed{\check{c'}}]{\mathsf{assign}} \mathtt{r}'\#c' \xrightarrow[\hat{c'}]{\mathsf{dispatch}[\_]} \cdots \tag{19}$$

where ℓ is store[i] (in passing a_i) or $\overline{\mathsf{load}[0]}$ (in retrieving a return value). Such a path, which starts from $\boxed{\hat{c}}$ and ends at $\boxed{\check{c'}}$, is called a dispatch path, which is valid if two conditions are met:

- DP-C1: $c = c'$ (implying that $\mathtt{r} = \mathtt{r}'$), and
- DP-C2: O is pointed by both $\mathtt{r}$ and $\mathtt{r}'$ (which are thus aliases) under exactly the same context.

However, L_{DC} can only ensure that $\mathtt{r}$ and $\mathtt{r}'$ are aliases but with no guarantee for the validity of this dispatch path. To filter out all L_{DC}-paths containing invalid dispatch paths, we use L_R given earlier to enforce `DP-C1` and `DP-C2`, thereby addressing CHL3 by restoring the callsite and context of $\mathtt{r}$. In particular, the siteRecovered-production enforces `DP-C1`, and the set of ctxRecovered-productions, together with the set of matched-productions, enforce `DP-C2`.

We have designed L_R to filter out all L_{DC}-paths containing invalid dispatch paths. Let us examine its productions by considering a generic dispatch path given in Eq. (19). The start symbol recoveredCtx would define a language that contains L_C if its alternative recoveredCtx siteRecovered were changed to recoveredCtx. Therefore, L_R comes into play only when a dispatch path is traversed by enforcing simply `DP-C1` and `DP-C2`.

To enforce `DP-C1`, the production siteRecovered $\longrightarrow$ $\boxed{\hat{c}}$ ctxRecovered $\boxed{\check{c}}$ states that if we start a dispatch process at a call site (flagged by $\boxed{\hat{c}}$), we must return to the same call site (flagged by $\boxed{\check{c}}$). For the dispatch path illustrated in Eq. (19), we are therefore guaranteed that $c = c'$, and consequently, $\mathtt{r} = \mathtt{r}'$. As a result, once $\boxed{\hat{c}}$ and $\boxed{\check{c}}$ are matched, c is recovered to appear at the ensuing dispatch edge so that dynamic dispatch can be performed at exactly the same callsite, i.e., c.

To enforce `DP-C2`, we rely on the ctxRecovered- and matched-productions, of which ctxRecovered $\longrightarrow$ $\check{c}$ ctxRecovered $\hat{c}$ plays the key role. Let us explain its theoretical basis by referring to a generic dispatch path in Eq. (19). We can express `DP-C2` equivalently as follows. Let $p_{\mathtt{r},O}$ be the $\overline{\text{flowsto}}$ path from $\mathtt{r}$ to O. Its inverse $\overline{p_{\mathtt{r},O}}$ is naturally a flowsto path. Let $p_{O,\mathtt{r}'}$ be the flowsto path from O to $\mathtt{r}'$. By Eq. (18), we obtain:

$$\begin{array}{rcl} \langle O, \mathscr{E}(\mathscr{B}({L_C}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}})))\rangle & \in & \mathsf{PTS}(\mathtt{r}, \mathscr{E}(\overline{\mathscr{B}({L_C}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))})) \\ \langle O, \mathscr{E}(\mathscr{B}({L_C}^{\mathsf{ex}}(p_{O,\mathtt{r}'})))\rangle & \in & \mathsf{PTS}(\mathtt{r}', \mathscr{E}(\overline{\mathscr{B}({L_C}^{\mathsf{en}}(p_{O,\mathtt{r}'}))})) \end{array} \tag{20}$$

As aliases, both $\mathtt{r}$ and $\mathtt{r}'$ must always point to O with exactly the same heap context:

$$\mathscr{E}(\mathscr{B}({L_C}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}}))) \quad = \quad \mathscr{E}(\mathscr{B}({L_C}^{\mathsf{ex}}(p_{O,\mathtt{r}'}))) \tag{21}$$

As a result, the entry contexts in $\overline{\mathscr{B}({L_C}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}}))}$ are fully balanced out by the exit contexts in $\mathscr{B}({L_C}^{\mathsf{ex}}(p_{O,\mathtt{r}'}))$ in L_C. Thus, the following must be true:

$$\mathscr{B}\Big(\overline{\mathscr{B}({L_C}^{\mathsf{ex}}(\overline{p_{\mathtt{r},O}}))}\mathscr{B}({L_C}^{\mathsf{ex}}(p_{O,\mathtt{r}'}))\Big) \quad = \quad \epsilon \tag{22}$$

Recall that exit and entry are inverses of each other according to L_C given in Eq. (2).

Now, both $\mathtt{r}$ and $\mathtt{r}'$ have exactly the same context (needed by `DP-C2`) iff the following holds:

$$\mathscr{E}(\overline{\mathscr{B}({L_C}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))}) \quad = \quad \mathscr{E}(\overline{\mathscr{B}({L_C}^{\mathsf{en}}(p_{O,\mathtt{r}'}))}) \tag{23}$$

In L_R, the exit contexts in $\overline{\mathscr{B}({L_C}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))}$ are thus needed to be balanced out by the entry contexts in $\mathscr{B}({L_C}^{\mathsf{en}}(p_{O,\mathtt{r}'})$ (in order to eliminate invalid dispatch paths):

$$\mathscr{B}\Big(\mathscr{B}({L_C}^{\mathsf{en}}(p_{O,\mathtt{r}'})\overline{\mathscr{B}({L_C}^{\mathsf{en}}(\overline{p_{\mathtt{r},O}}))}\Big) \quad = \quad \epsilon \tag{24}$$

We are now ready to explain the ctxRecovered- and matched- productions in L_R. When traversing a dispatch path illustrated in Eq. (19), ctxRecovered $\longrightarrow$ $\check{c}$ ctxRecovered $\hat{c}$ serves to enforce `DP-C2` according to Eq. (24), matched $\rightarrow$ siteRecovered is used to start traversing another dispatch path (recursively), the remaining productions serve to skip all matched contexts and all matched callsites. Informally, if we write down all the unmatched exit contexts we see when moving from $\mathtt{r}$ to O ($\mathtt{r}$ $\overline{\text{flowsto}}$ O) as $\check{c_1}, \ldots, \check{c_n}$, then all the unmatched entry contexts we see in returning from O to $\texttt{r}'$ (O flowsto $\texttt{r}'$) must be $\hat{c_n}, \ldots, \hat{c_1}$. ($\texttt{r} = \texttt{r}'$ due to DP-C1.)

To compute the points-to information according to L_{DCR}, we can continue to use Eq. (18) except that we only need to consider the L_{DCR}-paths in the PAG representation of the program.

Let us return to the two L_{DC}-paths given in Eq. (15) and Eq. (16) discussed at the beginning of Sec. 3.2.2. The L_{DC}-path in Eq. (15) is also an L_{DCR}-path since its dispatch paths are all valid. However, the L_{DC}-path in Eq. (16) is not an L_{DCR}-path, since its first dispatch path launched at callsite c3 starting from a and ending at a#c3 is not valid. Given that $\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\mathsf{a,A1}}}))} = \check{\mathsf{c}}\mathsf{1}$ and $\mathscr{B}(L_C{}^{\mathsf{en}}(p_{\mathsf{A1,a}})) = \hat{\mathsf{c}}\mathsf{2}$, which implies that $\mathscr{B}(\mathscr{B}(L_C{}^{\mathsf{en}}(p_{\mathsf{A1,a}}))\overline{\mathscr{B}(L_C{}^{\mathsf{en}}(\overline{p_{\mathsf{a,A1}}}))}) = \hat{\mathsf{c}}\mathsf{2}\check{\mathsf{c}}\mathsf{1} \neq \epsilon$, this dispatch path is invalid since $\check{\mathsf{c}}\mathsf{1}\hat{\mathsf{c}}\mathsf{2}$ cannot balance out according to ctxRecovered $\longrightarrow$ $\check{c}$ ctxRecovered $\hat{c}$.

Theorem 1. L_{DCR} is precise in handling parameter passing for virtual callsites.

Proof. Due to Lemmas 1 and 2, we only need to show by proceeding exactly as in the proof of Lemma 2 that for every virtual callsite "$\texttt{r.m}(a_1, \ldots, a_n)$; // c", where parameter passing for one of its arguments takes place under a given context C, L_{DCR} will perform its parameter passing for exactly the same set T of target methods found on the fly at this callsite under C by a separate call graph construction algorithm used in L_{FC}. This is true since L_R has succeeded in filtering out all and only L_{DC}-paths containing invalid dispatch paths (as argued above). □

We can now apply L_{DCR} to compute the points-to information in our motivating example (Fig. 3). Note that even though in its PAG (Fig. 7), the set of target methods at a virtual callsite is over-approximated conservatively by CHA [9], L_{DCR} will perform its own built-in on-the-fly call graph construction during its analysis, as illustrated in Eq. (7). Therefore, `C:foo()`, which appears in the PAG, will be filtered out due to on-the-fly call graph construction.

Similar to existing CFL-reachability formulations such as those proposed by Sridharan et al. [54, 55] and others [50, 63, 65], L_{DCR} can be effectively utilized for implementing demand-driven pointer analysis. It is important to note that whole-program and demand-driven pointer analyses are equivalent, with a slight operational difference. To analyze a program, whole-program analyses require a specified set M of entry methods, while demand-driven analyses require a specified set V of query variables (in the form of context and variable pairs). As a result, the whole-program analysis may not compute the points-to information for certain variables in V that are not part of the reachable code starting from the specified entry methods in M, unlike the demand-driven analysis.

Therefore, the precision of L_{DCR} is simply related to that of kCFA (which is specified by an Andersen-style formulation given in Fig. 1) as follows. Let $\mathsf{PTS}(v, c)$ be the points-to set of a pointer variable v computed by kCFA for a program (starting from `main()`), which implies that v is in reachable code from `main()` under context c. Then exactly the same points-to set $\mathsf{PTS}(v, c)$ can also be obtained for pointer v under context c from L_{DCR} according to Eq. (18). However, the converse is not true due to the existence of unreachable code when kCFA is applied. Consider a simple program given in Figure 11. If we choose `main()` as the only entry method for the program to be analyzed by a whole-program analysis, then `B.n()` becomes unreachable from `main()`. According to the rules specified in Fig. 1, the whole-program analysis will conclude that $\mathsf{PTS}(\mathsf{p}, [\mathsf{c1}]) = \{\langle \mathsf{O1}, [\,]\rangle\}$, indicating that p points to object O1 exclusively. However, when applying L_{DCR} for a demand-driven analysis to compute

```
static void main() {
  A a1 = new A(); // A1
  Object o1 = new Object(); // O1
  a1.m(o1); // c1
  a1.n();
}
class A {
  void m(Object p) {}
  void n() {}
}
class B extends A {
  void n() {
    A a2 = new A(); // A2
    Object o2 = new Object(); // O2
    a2.m(o2); // c2
}}
```

Fig. 11. An example for illustrating the equivalence and difference between L_{DCR} and kCFA.

the points-to information for `p`, we need to explicitly specify a context for the points-to query. There are three possibilities: `[c1]`, `[c2]`, and `[ ]`. According to Eq. (18), this results in $\mathsf{PTS}(\mathsf{p}, [\mathsf{c1}]) = \{\langle \mathsf{O1}, [\,] \rangle\}$, $\mathsf{PTS}(\mathsf{p}, [\mathsf{c2}]) = \{\langle \mathsf{O2}, [\,] \rangle\}$, and $\mathsf{PTS}(\mathsf{p}, [\,]) = \{\langle \mathsf{O1}, [\,] \rangle, \langle \mathsf{O2}, [\,] \rangle\}$, respectively. In the first scenario, the points-to information is computed for the callsite `c1`, resulting in the same outcome as the whole-program analysis. However, in the second scenario, the points-to information is computed specifically for the callsite `c2`, which is not reachable from the `main()` method. In the third scenario, the analysis takes into account both callsites, providing points-to information for each. It is important to highlight that in the second and third scenarios, the demand-driven analysis effectively treats `B.n()` as an additional entry method alongside `main()`. This example demonstrates the equivalence and disparity between whole-program and demand-driven analyses, highlighting the impact of unreachable code on the computed points-to information. Note that this asymmetric problem is also present when relating the precision of existing CFL-reachability formulations for supporting callsite-based context-sensitivity [50, 54, 63] and object-sensitivity [35, 37] to that of their corresponding Andersen-style inclusion-based formulations.

3.3 Time Complexities

In general, the traditional L_{FC}-reachability problem [54] is undecidable as it is the intersection of two CFLs interleaved with each other [47]. For a similar reason, the L_{DCR}-reachability problem is also undecidable as it is the intersection of three CFLs interleaved with each other. In fact, the $L_D \cap L_C$-, $L_D \cap L_R$-, and $L_C \cap L_R$-reachability problems are all undecidable. For example, the $L_C \cap L_R$-reachability problem is undecidable as the terminals in L_C (i.e., $\hat{c}$ and $\check{c}$) interleave with the terminals in L_R (i.e., $\hat{c}$, $\check{c}$, $\hat{\boxed{c}}$, and $\check{\boxed{c}}$), making $L_C \cap L_R$ context-sensitive. Intuitively, a string in $L_C \cap L_R$ can be understood as a path starting from a virtual callsite for the purposes of finding its receiver object and then returning back to the same virtual callsite. Since such paths can be different under different calling contexts for a given callsite, $L_C \cap L_R$ must be context-sensitive.

For a single CFL $L \in \{L_D, L_C, L_R\}$, the time complexity for solving its L-reachability problem is bounded by $O(m^3 n^3)$ from above, where m is its grammar size and n is the number of PAG nodes. As L_C is a standard Dyck-CFL defined over a PAG, which can be seen as a bidirected graph for L_C, the complexity for solving the L_C-reachability problem can be reduced to $O(p + n \cdot \alpha(n))$ [6], where p is the number of PAG edges, n is the number of PAG nodes, and $\alpha(n)$ is the inverse Ackermann function. For all practical purposes, we can apply k-limiting to L_C to make the L_{FC}-reachability problem computable in polynomial time again. Similarly, we can also apply k-limiting to both L_C and L_R to make the L_{DCR}-reachability problem computable in polynomial time again.

4 L_{DCR}: An Application

As the secondary contribution of this research, we demonstrate the utility of L_{DCR} by considering one significant application (among the list of potential applications discussed in Sec. 1). We introduce the first L_{DCR}-enabled pre-analysis, P3Ctx, for accelerating kCFA (implemented as a whole-program analysis in terms of the rules in Fig. 1) with selective context-sensitivity while always preserving its precision. This also serves to validate the correctness of L_{DCR}. In contrast, Selectx, a recently proposed pre-analysis developed based on L_{FC} [36], is not precision-preserving.

4.1 Selective Context-Sensitivity

Context-sensitivity is vital for enhancing pointer analysis precision. Blindly applying context sensitivity to all program variables and objects is time-consuming and provides limited precision benefits due to the presence of non-precision-critical elements. To address this, selective context-sensitivity focuses on applying context-sensitivity only to a pre-selected subset of precision-critical program variables and objects, while analyzing others in a context-insensitive manner. Several pre-analysis techniques have been proposed to support selective context-sensitive pointer analysis by pre-selecting a subset of program variables and objects [13, 14, 16, 20, 22, 30, 31, 36]. However, misclassification of precision-critical variables and objects in these techniques can lead to precision loss in the main pointer analysis. To address this issue, we introduce P3Ctx, the first precision-preserving pre-analysis technique based on L_{DCR} for identifying precision-critical variables and objects, supporting selective context-sensitivity in kCFA while maintaining analysis precision.

We have developed P3Ctx by following the same basic principle introduced in [36] for developing Selectx. For more technical details, we refer to [36].

4.1.1 CFL-Reachability-Guided Selections The basic idea in applying L_{FC} to develop Selectx [36] is simple. Let $p_{O,n,v}$ be a flowsto path operated by L_{FC} from some object O to some variable v, where n is a variable/object accessed in a method `M`. Let $p_{O,n}$ be its sub-path from O to n and $p_{n,v}$ its sub-path from n to v. Then n requires context-sensitivity (to prevent kCFA from potentially losing precision) only if the following three conditions are satisfied:

$$
\begin{aligned}
&\texttt{CS-C1} : L_F(p_{O,n,v}) \in L_F \\
&\texttt{CS-C2} : \wedge\, L_C(p_{O,n}) \in L_C \wedge L_C(p_{n,v}) \in L_C \\
&\texttt{CS-C3} : \wedge\, L_C^{\mathsf{en}}(p_{O,n}) \neq \epsilon \wedge L_C^{\mathsf{ex}}(p_{n,v}) \neq \epsilon
\end{aligned}
\tag{25}
$$

where L_C^{en} and L_C^{ex} are defined in Sec. 3.2.2. In this case, O from outside `M` flows into n along $p_{O,n}$ context-sensitively and n flows out of `M` into v along $p_{n,v}$ context-sensitively, via `M`'s parameters (or return variable) along each path. Note that $p_{O,n,v}$ itself is not required to be an L_{FC}-path.

Selectx will select n to be context-sensitive if `CS-C1`– `CS-C3` hold. By interpreting these conditions as being sufficient (rather than just necessary), Selectx is conservative as it may select some n to be context-sensitive even though kCFA loses no precision if it is analyzed context-insensitively.

However, Selectx may cause kCFA to lose precision. Consider our motivating example given in Fig. 3, for which whether `v` points to `O2` spuriously or not hinges on whether `d`, `o`, x, and D1 in bar() (containing a virtual callsite x.foo(d)) are analyzed context-sensitively or not. By reasoning about L_{FC} that operates on the PAG given in Fig. 4 for this example, Selectx will select all the four to be context-insensitive (causing v to point to O2), as none can flow out of bar() via its parameter x (which is also the receiver variable of x.foo(d)) in this PAG. Thus, CS-C3 fails to hold. In L_{FC}, which uses a separate algorithm for call graph construction, its PAG representation contains no dispatch paths that allow these four variables/objects to flow outside bar() via x as shown.

P3Ctx will always be precision-preserving as it leverages CS-C1– CS-C3 by substituting L_D for L_F, with L_{DC} operating on a new PAG representation including explicitly the dispatch paths for all virtual callsites in the program (as discussed below in Sec. 4.1.2). Consider our motivating example again, with its new PAG depicted in Fig. 7 for supporting built-in call graph construction. In L_D, parameter passing for d at x.foo(d) is CFL-reachability-related to its receiver variable x. Let $p_{\mathtt{O1},n,\mathtt{v}}$ be the path in Eq. (7) (which happens to be an L_{DC}-path). Let $n \in \{\mathtt{d}, \mathtt{o}, \mathtt{x}, \mathtt{D1}\}$. P3Ctx will select every n to be context-sensitive, since (1) $p_{\mathtt{O1},n,\mathtt{v}}$ is an L_D-path (CS-C1), (2) both $p_{\mathtt{O1},n}$ and $p_{n,\mathtt{v}}$ are L_C-paths (CS-C2), and (3) ${L_C}^{\mathsf{en}}(p_{\mathtt{O1},n}) = \hat{\mathtt{c1}} \neq \epsilon$ and ${L_C}^{\mathsf{ex}}(p_{n,\mathtt{v}}) = \check{\mathtt{c1}} \neq \epsilon$ (CS-C3).

4.1.2 Regularization To make P3Ctx as lightweight as possible so that we can efficiently make context-sensitivity selections without losing the performance benefits obtained from a subsequent main pointer analysis, we have decided to keep L_C unchanged as done in several earlier pre-analyses [35–37] but regularize L_D and L_R. We first regularize L_R to L_R^r as follows:

$$\mathsf{recoveredCtx} \longrightarrow \mathsf{recoveredCtx}\ \hat{c} \mid \mathsf{recoveredCtx}\ \check{c} \mid \mathsf{recoveredCtx}\ \boxed{\hat{\mathsf{c}}} \mid \mathsf{recoveredCtx}\ \boxed{\check{\mathsf{c}}} \mid \epsilon \tag{26}$$

As a result, we have $L_D \cap L_C \cap L_R^r = L_D \cap L_C = L_{DC}$. By noting further that the boxed edge labels in ${L_R}^r$ (i.e., $\boxed{\hat{\mathsf{c}}}$ and $\boxed{\check{\mathsf{c}}}$) are irrelevant to context-sensitivity selections and the regular entry/exit context labels in ${L_R}^r$ (i.e., $\hat{c}$ and $\check{c}$) have already been included in L_C, we conclude that ${L_R}^r$ (i.e., L_R) can be ignored safely (or conservatively). As $L_{DC} \supseteq L_{DCR}$ (i.e., L_{DC} captures all the possible value-flows that are captured by L_{DCR} for a given program) according to Lemma 2, it suffices to use L_{DC} in place of L_{FC} in Eq. (25) in developing our precision-preserving pre-analysis. Like the L_{FC}-reachability problem, the L_{DC}-reachability problem is also undecidable [47]. In this work, we follow [36] to first regularize L_D into L_{D^r}, and consequently, over-approximate L_{DC} to obtain $L_{D^rC} = L_{D^r} \cap L_C$. In Sec. 4.1.3, we will give an algorithm to verify CS-C1– CS-C3 efficiently by using L_{D^rC}.

We start with $L_0 = L_D$. We first over-approximate L_0 by disregarding its field-sensitivity requirement and thus obtain L_1 given below:

$$\begin{array}{rcl} \mathsf{flowsto} & \longrightarrow & \mathsf{new}\ (\mathsf{flows} \mid \overline{\mathsf{dispatch}})^* \\ \mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \mathsf{load} \\ \overline{\mathsf{flowsto}} & \longrightarrow & (\overline{\mathsf{dispatch}} \mid \overline{\mathsf{flows}})^*\ \overline{\mathsf{new}} \\ \overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}} \end{array} \tag{27}$$

In the absence of field-sensitivity, a dispatch ($\overline{\mathsf{dispatch}}$) edge behaves just like an assign ($\overline{\mathsf{assign}}$) edge and can thus be interpreted this way. As a result, we obtain L_2 below:

$$\begin{array}{rcl} \mathsf{flowsto} & \longrightarrow & \mathsf{new}\ \mathsf{flows}^* \\ \overline{\mathsf{flowsto}} & \longrightarrow & \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \mathsf{load} \\ \overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}} \end{array} \tag{28}$$

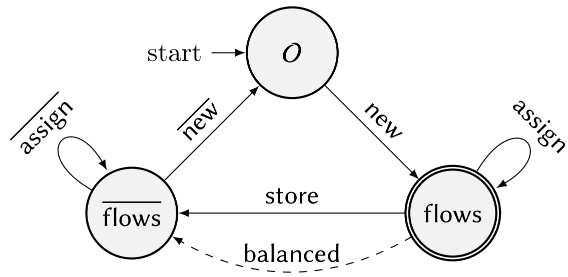


Fig. 12. A DFA for accepting L_{D^r}.

Our approximation goes further by treating a load ($\overline{\mathsf{load}}$) edge as also an assign ($\overline{\mathsf{assign}}$). As a result, we will no longer require a store ($\overline{\mathsf{load}}$) edge to be matched by a load ($\overline{\mathsf{store}}$) edge. This will give rise to L_3 below:

$$\begin{array}{rcl} \mathsf{flowsto} & \longrightarrow & \mathsf{new}\ \mathsf{flows}^* \\ \overline{\mathsf{flowsto}} & \longrightarrow & \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{flowsto}}\ \mathsf{flowsto} \\ \overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{flowsto}}\ \mathsf{flowsto}\ \overline{\mathsf{store}} \end{array} \tag{29}$$

Finally, we obtain $L_{D^r} = L_4$ given below by no longer distinguishing a store edge from its inverse, $\overline{\mathsf{store}}$ edge, so that we can represent both types of edges as a store edge:

$$\begin{array}{rcl} \mathsf{flowsto} & \longrightarrow & \mathsf{new}\ \mathsf{flows}^* \\ \overline{\mathsf{flowsto}} & \longrightarrow & \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}} \\ \mathsf{flows} & \longrightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathsf{assign}}^*\ \overline{\mathsf{new}}\ \mathsf{new} \\ \overline{\mathsf{flows}} & \longrightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{new}}\ \mathsf{new}\ \mathsf{assign}^*\ \mathsf{store} \end{array} \tag{30}$$

Lemma 3. $L_D \subseteq L_{D^r}$.

Proof. Follows from the fact that $L_i \subseteq L_{i+1}$. □

While L_{D^r} is identical to L_R regularized from L_F in Selectx [36], our PAG representation (Fig. 6), which makes all dynamic dispatch paths explicitly, differs fundamentally from the one operated by L_{FC} (Fig. 2). This ensures that P3Ctx is precision-preserving even though Selectx is not.

Let $G = (N, E)$ be the PAG of a program. We use Andersen's algorithm [1] instead of CHA [9] to build its call graph in order to sharpen the precision of P3Ctx.

We use a simple DFA shown in Fig. 12 designed to accept L_{D^r} exactly. P3Ctx runs interprocedurally in linear time of the number of the PAG edges in G. To deal with L_C, we make use of summary edges added into the PAG (facilitated by the dotted transition labeled as $\mathsf{balanced}$).

4.1.3 P3Ctx We follow [14] to develop a simple algorithm to verify `CS-C1`– `CS-C3` efficiently based on two properties that can be easily deduced from the DFA given in Fig. 12 as stated below.

Let $Q = \{O, \mathsf{flows}, \overline{\mathsf{flows}}\}$ be the set of states in the DFA and $\delta : Q \times \Sigma \mapsto Q$ be the underlying state transition function. Given a PAG edge $n_1 \xrightarrow{\ell} n_2 \in E$ in G with its state transition $\delta(q_1, \ell) = q_2$, we define $(n_1, q_1) \rightarrowtail (n_2, q_2)$ as a one-step transition. The transitive closure of $\rightarrowtail$, denoted by $\rightarrowtail^+$, represents a multiple-step transition. Any multiple-step transition from O to flows and that from $\overline{\mathsf{flows}}$ to O represent $\mathsf{flowsto}$ and $\overline{\mathsf{flowsto}}$ in Eq. (30), respectively. As $\mathsf{flowsto}$ and $\overline{\mathsf{flowsto}}$ in L_{D^r} are symmetric, the following two properties about this DFA are immediate:

- `PROP-O`. Let O be an object created in a method `M`. Then the following always holds:

$$\langle \mathtt{this}^{\mathtt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle O, O\rangle \iff \langle O, O\rangle \rightarrowtail^+ \langle \mathtt{this}^{\mathtt{M}}, \overline{\mathsf{flows}}\rangle$$

- `PROP-V`. Let v be a variable defined in a method `M`. Then the following always holds:

$$\langle \mathtt{this}^{\mathtt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle v, q\rangle \iff \langle v, \overline{q}\rangle \rightarrowtail^+ \langle \mathtt{this}^{\mathtt{M}}, \overline{\mathsf{flows}}\rangle$$

  where $q \in \{\mathsf{flows}, \overline{\mathsf{flows}}\}$ (since v is a variable).

To handle static callsites (static methods) uniformly as virtual callsites (virtual methods), we assume that a static callsite is invoked on a (unique) dummy receiver object. Thus, in our PAG representation for a program (constructed according to the rules given in Fig. 6), passing arguments and receiving return values for a method will all flow through its "`this`" variable.

P3Ctx can therefore verify `CS-C1`– `CS-C3` efficiently as follows. To verify `CS-C1` in Eq. (25), where L_F is now replaced by L_{D^r}, we do not have to start from an object to track its $\mathsf{flowsto}$ paths. For each method, we can start from its "`this`" variable by assuming reasonably and over-approximately that there always exists some object O that can flow into it. To verify `CS-C2`, we take advantage of summary edges as in [36] to verify the balanced-parentheses property in L_C-paths. To verify `CS-C3`, we check if there exists $q \in Q$ such that the following holds:

$$\langle \mathtt{this}^{\mathtt{M}}, \mathsf{flows}\rangle \rightarrowtail^+ \langle n, q\rangle \rightarrowtail^+ \langle \mathtt{this}^{\mathtt{M}}, \overline{\mathsf{flows}}\rangle \tag{31}$$

where `M` is the containing method of n. This implies that n lies on an L_{D^r}-path collecting some values coming from outside `M` via $\mathtt{this}^{\mathtt{M}}$ and pumping them out of `M` via $\mathtt{this}^{\mathtt{M}}$.

Let $R : Q \mapsto \wp(N)$ return the set of nodes in G reached at a state $q \in Q$. Then verifying `CS-C3`, i.e., checking Eq. (31) is equivalent to checking whether the following condition holds or not:

$$n \in R(O) \quad \vee \quad n \in R(\mathsf{flows}) \cap R(\overline{\mathsf{flows}}) \tag{32}$$

The first disjunct says that if an object n is in $R(O)$, then Eq. (31) holds due to `PROP-O`. Its second disjunct says that if a variable n is in $R(\mathsf{flows}) \cap R(\overline{\mathsf{flows}})$, then Eq. (31) holds (due to `PROP-V`).

Fig. 13 gives our algorithm for performing our P3Ctx pre-analysis in terms of three rules. Essentially, P3Ctx computes R by conducting a simple inter-procedural reachability analysis in G. In Fig. 13, $R^{-1} : N \mapsto \wp(Q)$, which returns the set of reachable states for a node in G, is the inverse of R. For the three rules, [F-Init] does the initializations as needed, [F-Propa] computes the reachable states for each node iteratively, and finally, [F-Sum] performs a standard context-sensitive summary for a callsite invoking `M` [48] by adding a summary edge $n_1 \xrightarrow{\text{balanced}} n_2$ in G to capture inter-procedural reachability across the callsite (to avoid re-computing the same reachability information unnecessarily for the same callsite).

Theorem 2. *kCFA* (performed in terms of the rules in Fig. 1) produces exactly the same points-to information when performed with selective context-sensitivity under P3Ctx.

$$\frac{n_1 \xrightarrow[\hat{c}]{-} \mathtt{this}^{\mathtt{M}} \in E}{\mathtt{this}^{\mathtt{M}} \in R(\mathsf{flows}) \quad \mathsf{flows} \in R^{-1}(\mathtt{this}^{\mathtt{M}})} \quad \text{[F-INIT]}$$

$$\frac{n_1 \xrightarrow{\ell} n_2 \in E \quad q_1 \in R^{-1}(n_1) \quad \delta(q_1, \ell) = q_2}{n_2 \in R(q_2) \quad q_2 \in R^{-1}(n_2)} \quad \text{[F-PROPA]}$$

$$\frac{n_1 \xrightarrow[\hat{c}]{-} \mathtt{this}^{\mathtt{M}} \in E \quad \mathtt{this}^{\mathtt{M}} \xrightarrow[\check{c}]{-} n_2 \in E \quad \overline{\mathsf{flows}} \in R^{-1}(\mathtt{this}^{\mathtt{M}})}{n_1 \xrightarrow{\mathsf{balanced}} n_2 \in E} \quad \text{[F-SUM]}$$

Fig. 13. Rules for conducting P3CTX over $G = (N, E)$.

Proof. Follows from the facts that (1) Eq. (25) provides a set of necessary conditions for supporting selective context-sensitivity, (2) L_{DCR} provides a specification of kCFA with built-in on-the-fly callgraph construction via CFL-reachability, (3) $L_{D^rC} \supseteq L_{DC} \supseteq L_{DCR}$ (Lemma 3), and (4) [F-INIT] has weakened CS-C1 by starting from the `this` variable of every method instead of every object O. □

The worst-case time complexity of P3Ctx in analyzing a program on $G = (N, E)$ is $O(|E| \times |Q|)$, which is linear to $|E|$, where $|Q| = 3$ is the number of states in our DFA.

4.2 Evaluation

The primary focus of this work lies in the development L_{DCR} as the first CFL-reachability specification of kCFA with its own built-in call graph construction. In order to demonstrate its utility, we have also developed P3Ctx, the first precision-preserving pre-analysis for accelerating kCFA (Theorem 2). In this section, we provide an experimental validation of this claim on P3Ctx and compare it with two non-precision-preserving state-of-the-art pre-analyses, Selectx [36] and Zipper [30]. Our experimental results show that P3Ctx represents a new advance with better efficiency-precision trade-offs in a number of application scenarios highlighted below.

4.2.1 Experimental Setup We have implemented kCFA (i.e., Andersen's inclusion-based formulation given in Fig. 1) and P3Ctx (Fig. 13) in Soot [60] on top of its context-insensitive Andersen's pointer analysis, Spark [28], which is used for building the PAG of a program (including its call graph) for P3Ctx. To compare P3Ctx with Selectx and Zipper, we have reused their implementations provided in the Selectx artifact[1]. For evaluation purposes, we have followed a few common practices adopted in the pointer analysis literature [14, 16, 35–37, 44, 57]. We use a reflection log generated by a dynamic reflection analysis tool, TamiFlex [4] for resolving Java reflection. For native code, we use the method summaries provided in Soot. String factory objects and exception-like objects are distinguished per dynamic type and analyzed context-insensitively.

We have selected a set of 13 benchmarks from the DaCapo benchmark suite (the latest version `6cf0380`) together with a large Java library (`JRE1.8.0_31`). We have excluded only `jython` as all pointer analyses (evaluated in this section) except Spark cannot analyze this

[1]Selectx artifact is available at https://doi.org/10.5281/zenodo.4732680

benchmark to completion under a time budget of 12 hours due to its overly conservative reflection log [59].

We have carried out all our experiments on an Intel(R) Xeon(R) W-2245 3.90GHz machine with 512GB of RAM, running on Ubuntu 20.04.3 LTS (Focal Fossa).

4.2.2 Results Table 3 contains the results for kCFA, P-kCFA (i.e., kCFA accelerated by P3Ctx), S-kCFA (i.e., kCFA accelerated by Selectx), and Z-kCFA (i.e., kCFA accelerated by Zipper), where $k \in \{1, 2\}$. The results for Spark are also included for comparison purposes. Note that for $k \geqslant 3$, kCFA is unscalable for all the 13 programs under a time budget of 12 hours and thus has never been considered in the pointer analysis literature [20, 30, 31, 36, 44, 52, 57, 59].

Precision We measure the precision of a pointer analysis by considering four commonly used metrics [11, 14, 30, 37, 52, 57]: (1) "#Call Edges": the number of call graph edges discovered, (2) "#Fail Casts": the number of type casts that may fail, (3) "#Alias Pairs": the number of base variable pairs of stores and loads that are queried to be may-alias with trivial must aliases (e.g., due to direct assignments) being excluded [11], and (4) "Avg PTS": the average number of objects pointed by a variable by considering only the local variables in the Java methods (being analyzed). For each of the four precision metrics, smaller is better.

For each metric M, M_{PTA} denotes the result obtained by PTA, where PTA denotes any pointer analysis in {Spark, kCFA, P-kCFA, S-kCFA, Z-kCFA }. Let A-kCFA $\in \{P\text{-}k\mathsf{CFA}, S\text{-}k\mathsf{CFA}, Z\text{-}k\mathsf{CFA}\}$ be one of the three variants of kCFA such that A-kCFA is no less precise than Spark but no more precise than kCFA. We define the precision loss of A-kCFA with respect to kCFA on metric M as:

$$\Delta^{M}_{A\text{-}k\mathsf{CFA}} = \frac{(M_{\text{Spark}} - M_{k\mathsf{CFA}}) - (M_{\text{Spark}} - M_{A\text{-}k\mathsf{CFA}})}{M_{\text{Spark}} - M_{k\mathsf{CFA}}} = \frac{M_{A\text{-}k\mathsf{CFA}} - M_{k\mathsf{CFA}}}{M_{\text{Spark}} - M_{k\mathsf{CFA}}} \tag{33}$$

The precision improvement going from Spark to kCFA (considered as the baseline) is regarded as 100%. If $M_{A\text{-}k\mathsf{CFA}} = M_{k\mathsf{CFA}}$ (i.e., A-kCFA is equally precise as kCFA), then $\Delta^{M}_{A\text{-}k\mathsf{CFA}} = 0\%$, implying that A-kCFA loses no precision at all. On the other hand, if $M_{A\text{-}k\mathsf{CFA}} = M_{\text{Spark}}$ (i.e., A-kCFA degenerates into Spark), then $\Delta^{M}_{A\text{-}k\mathsf{CFA}} = 100\%$, implying that A-kCFA loses all the precision gained by kCFA.

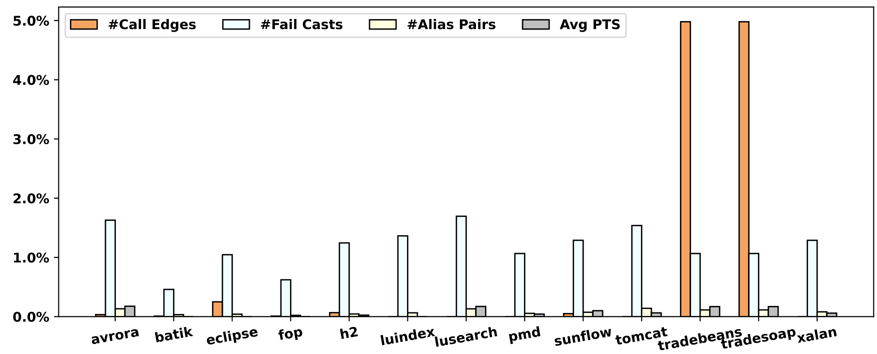
For all the 13 benchmarks considered, P-kCFA is always precision-preserving (i.e., for each precision metric M considered, $\Delta^{M}_{P\text{-}k\mathsf{CFA}} = 0\%$ always holds) as proved in Theorem 2 and validated further in Table 3 with P-kCFA producing exactly the same points-to results as kCFA.

Fig. 14 plots the precision loss of S-2CFA and Z-2CFA. As shown in Fig. 14a, S-2CFA suffers from only a small loss of precision (0.8%, 1.2%, 0.1% and 0.1% for "#Call Edges", "#Fail Casts", "#Alias Pairs" and "Avg PTS", respectively, on average) as Selectx exploits L_{FC} [54] for making its context-sensitivity selections. We will introduce two examples below to explain the relatively large precision loss observed in `tradebeans`, `tradesoap` and `eclipse`. On the other hand, as shown in Fig. 14b, Z-2CFA suffers from a higher precision loss as Zipper exploits pattern-based heuristics to make its context-sensitivity selections. On average, its precision loss percentages for "#Call Edges", "#Fail Casts", "#Alias Pairs", and "Avg PTS" are 6.2%, 8.1%, 2.2%, and 2.0%, respectively.

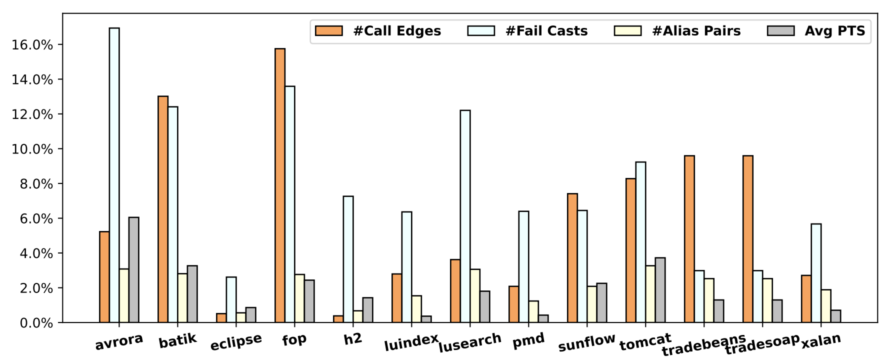
Fig. 15 gives an example for illustrating why S-2CFA loses precision in `tradebeans` and `tradesoap`. Unlike P-2CFA, S-2CFA fails to identify the call in line 15 as a monomorphic call to `toString()` defined in `java.lang.String`. This example includes three classes with `TreeMap`

Table 3. The precision and efficiency of kCFA, P-kCFA (kCFA accelerated by P3Ctx), S-kCFA (kCFA accelerated by Selectx), and Z-kCFA (kCFA accelerated by Zipper). The results for Spark are also included. For each main analysis in $A \in \{P\text{-}k\text{CFA}, S\text{-}k\text{CFA}, Z\text{-}k\text{CFA}\}$, the analysis times are given as $x(y)$, where x is the analysis time of A and y is the corresponding pre-analysis time (in seconds). For all metrics, smaller is better.

Program	Metrics	Spark	1CFA	P-1CFA	S-1CFA	Z-1CFA	2CFA	P-2CFA	S-2CFA	Z-2CI
avrora	Time(secs)	6.6	18.0	4.7 (1.2)	3.1 (21.5)	2.8 (4)	577.1	142.5 (1.2)	16.8 (21.6)	11.2 (
	#Call Edges	57509	55267	55267	55267	55403	54505	54505	54506	546
	#Fail Casts	1197	931	931	931	965	890	890	895	9
	#Alias Pairs	22327	13700	13700	13700	13703	13268	13268	13280	135
	Avg PTS	36.19	25.87	25.87	25.87	26.48	24.78	24.78	24.80	25.
batik	Time(secs)	30.9	81.0	28.0 (4.7)	25.3 (169.5)	23.1 (243)	1473.9	466.5 (4.8)	271.1 (174.4)	276.5 (23
	#Call Edges	171409	151995	151995	151997	152025	147428	147428	147430	1505
	#Fail Casts	4573	3709	3709	3709	3713	3485	3485	3490	36
	#Alias Pairs	68130	38005	38005	38005	38012	32288	32288	32300	332
	Avg PTS	114.43	71.67	71.67	71.67	71.71	66.65	66.65	66.65	68.
eclipse	Time(secs)	14.8	48.7	23.3 (2.0)	20.1 (54.6)	19.7 (14)	1221.1	331.0 (2.0)	171.8 (56.8)	143.9 (1
	#Call Edges	110089	97960	97960	98000	98052	93662	93662	93703	937
	#Fail Casts	2896	2470	2470	2471	2474	2322	2322	2328	23
	#Alias Pairs	107389	58489	58489	58500	58504	51404	51404	51427	517
	Avg PTS	101.12	63.49	63.49	63.47	63.80	59.28	59.28	59.26	59.
fop	Time(secs)	76.0	318.8	123.1 (10.6)	113.1 (603.8)	104.0 (355)	6019.6	2399.6 (10.8)	1901.7 (604.5)	1405.1 (35
	#Call Edges	358738	325547	325547	325551	325591	313954	313954	313958	3210
	#Fail Casts	9057	8226	8226	8228	8239	7931	7931	7938	80
	#Alias Pairs	323628	277047	277047	277047	277065	267389	267389	267401	2689
	Avg PTS	233.48	141.19	141.19	141.19	141.25	132.98	132.98	132.98	135.
h2	Time(secs)	16.1	75.7	18.5 (2.9)	15.8 (74.1)	14.3 (40)	6406.8	4164.6 (2.8)	3807.8 (74.4)	3127.4 (3
	#Call Edges	144711	135775	135775	135782	135806	134234	134234	134241	1342
	#Fail Casts	2880	2477	2477	2477	2482	2398	2398	2404	24
	#Alias Pairs	77978	39209	39209	39209	39236	33331	33331	33351	336
	Avg PTS	72.61	34.61	34.61	34.61	34.68	32.63	32.63	32.64	33.
luindex	Time(secs)	18.5	41.0	24.0 (1.9)	22.6 (48.1)	20.1 (8)	829.1	232.3 (1.9)	109.0 (48.2)	82.3 (
	#Call Edges	85850	79431	79431	79431	79602	78190	78190	78190	784
	#Fail Casts	1726	1359	1359	1360	1376	1286	1286	1292	13
	#Alias Pairs	50530	32905	32905	32905	32908	31795	31795	31807	320
	Avg PTS	53.10	24.75	24.75	24.75	24.87	23.04	23.04	23.04	23.
lusearch	Time(secs)	5.3	12.6	3.5 (1.0)	2.3 (13.9)	1.9 (3)	414.0	129.3 (1.0)	9.6 (13.9)	7.1 (
	#Call Edges	45285	43117	43117	43117	43198	42412	42412	42412	425
	#Fail Casts	955	702	702	702	719	660	660	665	6
	#Alias Pairs	20382	11693	11693	11693	11696	11263	11263	11275	115
	Avg PTS	31.38	20.73	20.73	20.74	20.85	19.73	19.73	19.75	19.
pmd	Time(secs)	20.3	109.5	42.6 (3.0)	37.2 (139.1)	35.9 (25)	16006.8	13715.8 (3.0)	13671.4 (139.1)	9356.3 (2
	#Call Edges	159395	153150	153150	153150	153387	152090	152090	152090	1522
	#Fail Casts	4702	4321	4321	4321	4325	4233	4233	4238	42
	#Alias Pairs	114914	95977	95977	95977	95979	93083	93083	93095	933
	Avg PTS	90.97	68.76	68.76	68.76	68.79	67.48	67.48	67.49	67.
sunflow	Time(secs)	9.9	25.9	7.4 (1.8)	5.5 (46.4)	5.3 (9)	643.1	165.1 (1.7)	33.0 (45.9)	27.7 (
	#Call Edges	77346	74198	74198	74200	74241	73392	73392	73394	736
	#Fail Casts	2192	1771	1771	1773	1776	1649	1649	1656	16
	#Alias Pairs	36952	21670	21670	21670	21678	20703	20703	20715	210
	Avg PTS	51.31	33.62	33.62	33.62	33.69	31.34	31.34	31.36	31.
tomcat	Time(secs)	7.4	18.9	5.8 (1.3)	4.0 (20.8)	3.7 (4)	632.9	148.7 (1.3)	16.1 (20.8)	11.7 (
	#Call Edges	60649	57933	57933	57933	58024	57073	57073	57073	573
	#Fail Casts	1264	959	959	960	963	874	874	880	9
	#Alias Pairs	30775	24504	24504	24504	24507	22202	22202	22214	224
	Avg PTS	39.88	25.37	25.37	25.37	25.51	24.03	24.03	24.04	24.
tradebeans	Time(secs)	8.7	25.9	7.6 (1.5)	5.6 (41.7)	5.2 (9)	737.4	166.5 (1.5)	30.2 (43.4)	18.2 (
	#Call Edges	70911	67742	67742	67742	67858	66814	66814	67018	672
	#Fail Casts	1523	1132	1132	1132	1135	1054	1054	1059	10
	#Alias Pairs	36256	27175	27175	27175	27178	25683	25683	25695	259
	Avg PTS	47.67	31.80	31.80	31.80	31.87	29.95	29.95	29.98	30.
tradesoap	Time(secs)	8.4	24.8	7.7 (1.6)	5.8 (46.8)	5.2 (9)	703.0	162.8 (1.5)	29.9 (49.4)	17.9 (
	#Call Edges	70911	67742	67742	67742	67858	66814	66814	67018	672
	#Fail Casts	1523	1132	1132	1132	1135	1054	1054	1059	10
	#Alias Pairs	36256	27175	27175	27175	27178	25683	25683	25695	259
	Avg PTS	47.67	31.80	31.80	31.80	31.87	29.95	29.95	29.98	30.
xalan	Time(secs)	8.5	27.3	7.4 (1.4)	5.5 (42.6)	5.0 (16)	702.8	162.3 (1.6)	34.2 (42.3)	26.0 (1
	#Call Edges	69608	67132	67132	67132	67210	66360	66360	66360	664
	#Fail Casts	1807	1473	1473	1473	1477	1419	1419	1424	14
	#Alias Pairs	42119	28280	28280	28280	28283	27259	27259	27271	275
	Avg PTS	45.29	29.41	29.41	29.41	29.47	28.29	28.29	28.30	28.



(a) Precision loss of S-2CFA



(b) Precision loss of Z-2CFA

Fig. 14. The precision loss of S-2CFA and Z-2CFA (computed according to Eq. (33)).

and
`CaseInsensitiveComparator` abstracted from `JDK8` and `StringComparator` abstracted from `tradebeans`. In `main()`, two `TreeMap` objects are created and used as the receiver objects to invoke `put()` with an integer object and a string object as its first argument, respectively (lines 26-27). When `put()` is invoked on each `TreeMap` object, a virtual call `compare()` is invoked on the `comparator` object stored in the `TreeMap` object. When 2CFA is applied, `put()` will be analyzed under two calling contexts, [`c1`] and [`c2`]. Under [`c1`], `cmp` points to `O1` and `k` points to `O5`, so that `compare()` defined in line 10 is identified to be called under [`c3`, `c1`] in line 6. Under [`c2`], `cmp` points to `O2` and `k` points to `O6`, so that `compare()` defined in line 14 is called under [`c3`, `c2`] in line 6, in which case, `o1` points to `O6` uniquely in line 14. As a result, the virtual call made in line 15 invokes only the `toString()` method defined in `java.lang.String`. Selectx relies on L_{FC} [54] to make its context-sensitivity selections and will identify `cmp` and `k` in `put()` to be context-insensitive since `CS-C3` in Eq. (25) is not satisfied with respect to L_{FC}. When S-2CFA is applied, `o1` is found to point to both `O5` and `O6` conservatively even under [`c3`, `c2`], making the call in line 15 polymorphic incorrectly (with its two call targets being `toString()` in `java.lang.String` and `toString()`

in java.lang.Integer). In contrast, P3Ctx makes its context-sensitivity selections based on L_{DCR}, concluding that cmp and k in put() should be context-sensitive since CS-C3 in Eq. (25) is satisfied with respect to L_{DCR}. When P-2CFA is applied, o1 is found to point to 06 uniquely under context [c3, c2], making the call in line 15 monomorphic correctly. When analyzing tradebeans and tradesoap, S-2CFA suffers from a precision loss of about 5% for "#Call Edges" this way, which can be undesirable for many precision-critical client analyses such as software security analysis.

Fig. 16 gives another example abstracted from eclipse and JDK8 to further illustrate why S-2CFA can lose precision. The situation for S-1CFA is similar. In line 23 (26) of main(), we call execute() with 02 (05) as the receiver object and 04 (06) as its argument. As Selectx will select all the variables in execute() and reject() to be context-insensitive (for the similar reasons discussed in the previous example), S-2CFA will conclude that $\mathsf{PTS}(\mathsf{r}, [\,]) = \{\langle 04, [\,]\rangle, \langle 06, [\,]\rangle\}$, making the call to run() in line 6 polymorphic. In contrast, P3Ctx will select all the variables in execute() and reject() to be context-sensitive. When P-2CFA is applied, reject() has two calling contexts, [c3, c1] and [c3, c2]. Under [c3, c1], we find that $\mathsf{PTS}(\mathsf{this}^{\mathsf{reject}}, [\mathsf{c3}, \mathsf{c1}]) = \{\langle 02, [\,]\rangle\}$, $\mathsf{PTS}(02.\mathsf{handler}, [\,]) = \{\langle 01, [\,]\rangle\}$, and $\mathsf{PTS}(\mathsf{r}, [\mathsf{c4}, \mathsf{c3}]) = \mathsf{PTS}(\mathsf{p}, [\mathsf{c3}, \mathsf{c1}]) = \{\langle 04, [\,]\rangle\}$. Under [c3, c2], we find that $\mathsf{PTS}(\mathsf{this}^{\mathsf{reject}}, [\mathsf{c3}, \mathsf{c2}]) = \{\langle 05, [\,]\rangle\}$ and $\mathsf{PTS}(05.\mathsf{handler}, [\,]) = \emptyset$, and in addition, whatever p points to cannot be passed to r. By combining these two cases, P-2CFA concludes that r only points to 04, and consequently, the call to run() in line 6 is monomorphic. When analyzing eclipse, S-2CFA introduces dozens of such spurious call edges.

Efficiency We measure the efficiency of a pointer analysis by the time elapsed in analyzing a program (as an average of three runs). For kCFA, this will simply be the time that kCFA spends on analyzing a program. Its three variants, P-kCFA, S-kCFA and Z-kCFA, are obtained according to P3Ctx, Selectx and Zipper, respectively. For each variant, its efficiency in analyzing a program is computed as the sum of its analysis time and its corresponding pre-analysis time. The analysis time of Spark is ignored since the context-insensitive points-to information produced by Spark is shared by all three pre-analyses. For each variant of kCFA, A−kCFA, where $A \in \{P, S, Z\}$, we include its corresponding pre-analysis time in both A−1CFA and A−2CFA in order to model practical scenarios where a software application is usually analyzed by A−kCFA for one fixed value of k, which is determined based on some particular efficiency-precision trade-offs made for the application.

Table 3 gives the analysis times for all the pointer analyses evaluated. For each variant of kCFA, P-kCFA, S-kCFA or Z-kCFA, we run its corresponding pre-analysis separately when $k = 1$ and $k = 2$. Therefore, for the same program, the two pre-analysis times may differ slightly.

Fig. 17 plots the speedups of P-kCFA, S-kCFA, and Z-kCFA over kCFA for all the 13 benchmarks considered, where $k \in \{1, 2\}$. In general, all three pre-analyses, P3Ctx, Selectx, and Zipper, can speed up kCFA substantially for $k = 2$. As shown in Fig. 17a, Z-2CFA achieves the highest speedups, ranging from 41.0× (for lusearch) to 1.7× (for pmd) with an average of 10.9×. S-2CFA's speedups range from 17.6× (for lusearch) to 1.2× (for pmd) with an average of 6.0×. Finally, P3Ctx achieves the lowest speedups, ranging from 4.4× (for tradebeans) to 1.2× (for pmd) with an average of 3.2×. When $k = 1$, the situation has been revered (as the main analysis 1CFA takes significantly less time to complete than 2CFA and the pre-analysis overhead added by P3Ctx on top of 1CFA is relatively small). As shown in Fig. 17b, only P3Ctx can improve the performance of 1CFA substantially. Zipper can speed up 1CFA for most programs, but fares more poorly than P3Ctx overall. On the other hand, Selectx causes

```
class TreeMap {
  Comparator comparator;
  TreeMap(Comparator cmp1) { this.comparator = cmp1; }
  void put(Object k, Object v) {
    Comparator cmp = this.comparator;
    int i = cmp.compare(k, ...); // c3
}}
// in java.lang.String
class CaseInsensitiveComparator implements Comparator {
  int compare(String p1, String p2) { return 0; }
}
// in org.apache.geronimo.main
class StringComparator implements Comparator {
  int compare(Object o1, Object o2) {
    String s1 = o1.toString(); // #Call Edges?
    return s1.compareTo(o2.toString());
}}
void main() {
  Comparator cmp1 = new CaseInsensitiveComparator(); // O1
  Comparator cmp2 = new StringComparator(); // O2
  TreeMap map1 = new TreeMap(cmp1); // O3
  TreeMap map2 = new TreeMap(cmp2); // O4
  Integer x = new Integer(1); // O5
  String y = new String(); // O6
  z = new String(); // O7
  map1.put(x, z); // c1
  map2.put(y, z); // c2
}
```

Fig. 15. An example abstracted from tradebeans and JDK8 to illustrate why SELECTX is not precision-preserving (by applying L_{FC} to determine precision-critical variables/objects in a program).

1CFA to run more slowly when its pre-analysis overhead is accounted for. To summarize, P-1CFA achieves the highest speedups, ranging from 3.5× (for `h2`) to 1.6× (for `luindex`) with an average of 2.6×, Z-1CFA's speedups range from 2.6× (for `avrora` and `lusearch`) to 0.3× (for `batik`) with an average of 1.5×, and finally, S-1CFA has the lowest speedups, ranging from 0.8× (for `h2` and `tomcat`) to 0.4× (for `batik` and `fop`) with an average of 0.6×.

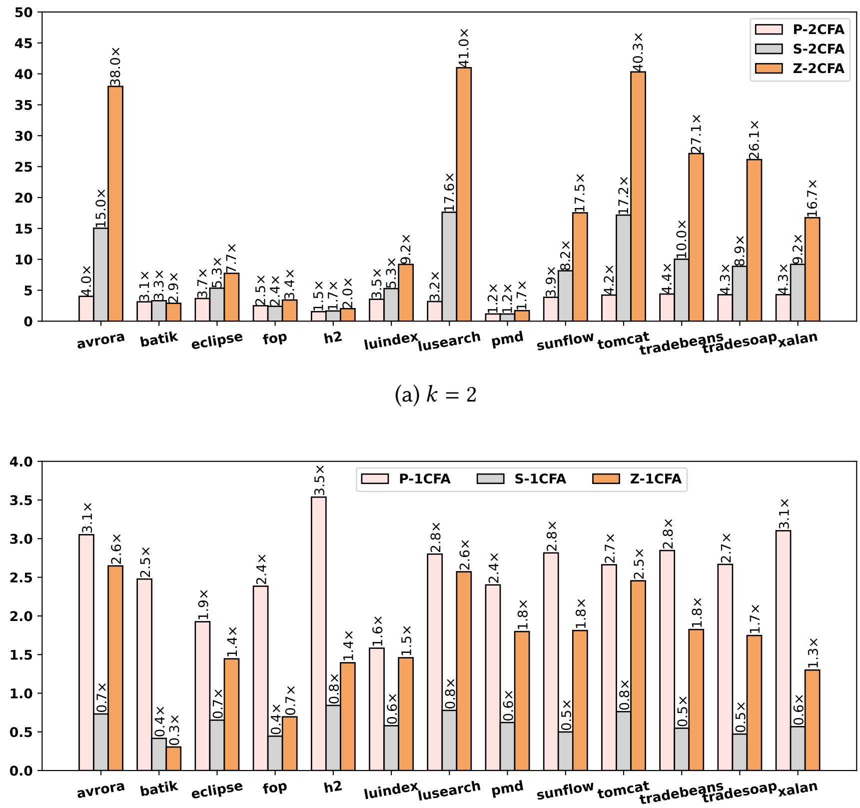
When considering both the precision and efficiency achieved by P-kCFA, S-kCFA and Z-kCFA, we can draw several conclusions. First, for precision-critical client analyses such as software security analysis, P-kCFA is recommended since it not only runs significantly faster than the baseline kCFA but also preserves its precision. Second, for certain client analyses that expect to use a pointer analysis that has the precision of 1CFA but runs as fast as possible, P-1CFA is recommended as it is faster than S-1CFA and Z-1CFA while achieving the same precision as 1CFA. Finally, for certain client analyses that expect to use a pointer analysis with the precision of 2CFA, our recommendation is Z-2CFA if these clients prioritize efficiency over precision (by trading willingly some loss of precision for efficiency gains), or S-2CFA if these clients still prioritize efficiency over precision but can only accept a negligible

```
// in org.eclipse.osgi.internal.framework
class EquinoxContainerAdaptor {
  Callable<Executor> createLazyExecutorCreator() {
  return new Callable<Executor>() { Executor call() {
    RejectedExecutionHandler handler = new RejectedExecutionHandler() { // O1
      void rejectedExecution(Runnable r, ThreadPoolExecutor exe) { r.run(); }};
    return new ThreadPoolExecutor(handler); // O2
    }};
}}
class ThreadPoolExecutor implements Executor {
  RejectedExecutionHandler handler;
  ThreadPoolExecutor(RejectedExecutionHandler h) { this.handler = h; }
  void reject(Runnable p) { this.handler.rejectedExecution(p, this); } // c4
  void execute(Runnable q) { this.reject(q); // c3
}}
class FutureTask implements Runnable { void run() { ... } }
class ScheduledFutureTask extends FutureTask { void run() { ... } }
void main () {
  EquinoxContainerAdaptor adaptor = new EquinoxContainerAdaptor(); // O3
  Callable<Executor> callable = adaptor.createLazyExecutorCreator();
  Executor executor1 = callable.call();
  FutureTask t1 = new FutureTask(); // O4
  executor1.execute(t1); // c1
  Executor executor2 = new ThreadPoolExecutor(null); // O5
  FutureTask t2 = new ScheduledFutureTask(); // O6
  executor2.execute(t2); // c2
}
```

Fig. 16. Another example abstracted from eclipse and JDK8 to illustrate why SelectX loses precision.

loss of precision, or P-2CFA otherwise (i.e., if these clients prioritize precision over efficiency but still expect the pointer analysis to run as fast as possible).

Discussion The analysis times of kCFA, P-kCFA, S-kCFA, and Z-kCFA, and consequently, the speedups achieved by P-kCFA, S-kCFA, and Z-kCFA over kCFA, may depend on the pointer analysis frameworks in which kCFA, P-kCFA, S-kCFA, and Z-kCFA are implemented and the benchmarks selected during the evaluation. In this work, we have conducted our evaluation in a popular pointer analysis framework Soot by using all the benchmarks (except `jython`) from the latest DaCapo benchmark suite and comparing kCFA, P-kCFA, S-kCFA, and Z-kCFA under the same standard experimental setting used in the pointer analysis literature [14, 30, 57]. Our evaluation shows that L_{DCR} can enable P-kCFA to run faster than kCFA without any precision loss. In addition, our evaluation also shows that P-kCFA represents a new alternative to S-kCFA and Z-kCFA, advancing the state of the art in accelerating kCFA with selective context-sensitivity.



(a) $k = 2$

(b) $k = 1$

Fig. 17. The speedups of *P*-*k*CFA, *S*-*k*CFA, and *Z*-*k*CFA over *k*CFA based on the analysis times in Table 3.

5 Related Work

We discuss only the prior work closely related to this work, by focusing on (1) exploiting CFL-reachability in developing context-sensitive pointer analysis algorithms for object-oriented languages and (2) selective context-sensitivity for accelerating such pointer analysis algorithms.

5.1 CFL-Reachability

In program analysis, CFL-reachability [46, 48] was initially introduced for supporting inter-procedural dataflow analysis. It has since been used in tacking many other problems such as pointer analysis [35, 37, 50, 54, 55, 62–65], information flow [32, 39], and type inference [43, 45].

For callsite-based context-sensitive pointer analysis [50, 54, 63], the CFL-reachability formulation used so far relies on a separate mechanism for call graph construction (in advance or on the fly), as discussed in Sec. 2.2. In this paper, we introduce a CFL-reachability formulation with such a mechanism built in, by using a new language L_{DCR} expressed as the intersection of three CFLs.

An earlier attempt to address the same problem was proposed by Sridharan in his PhD thesis [53], which, to the best of our knowledge, has not been published in a peer-reviewed publication. In his approach, context-sensitive pointer analysis with on-the-fly call graph construction is formulated as the intersection of two CFLs, $L_{OTF} \cap L_{C'}$, on a specially designed PAG. L_{OTF}, which extends L_F to support on-the-fly call graph construction, is quite similar to L_D proposed in this paper. At a virtual callsite, both L_{OTF} and L_D first establish a $\overline{\textsf{flowsto}}$ value-flow to find its receiver objects, together with their types, then find a $\textsf{flowsto}$ value-flow to return to the virtual callsite, and finally, perform the virtual call dispatch according to the types of the discovered receiver objects. However, L_{OTF} and L_D differ mainly in how they handle parameter passing and method returns at a virtual callsite. In L_{OTF}, the parameter passing and method returns are modeled essentially as `assign` edges with different edge labels such as `receiver[i][s]`, `paramForType[T][s]` and `returnForType[T][s]`, making its grammar intuitive but somewhat complex. In contrast, L_D handles parameter passing and method returns uniformly as stores and loads, making its grammar also intuitive yet simple. In addition, L_{OTF} requires statement matching in its non-terminals such as `dispatch[i][s]` to ensure that the parameter passing for an argument to a parameter always happens at the same callsite. This is achieved in our L_R language by using the boxed edge labels as revealed in Eq. (17). $L_{C'}$ in Sridharan's formulation is identical to L_C despite some notational differences. Finally, $L_{OTF} \cap L_{C'}$ is sound but less precise than L_{DCR} due to the lack of L_R introduced in this paper. Without L_R, a context used for performing parameter passing at a virtual callsite can be restored incorrectly as a different context after finding the dispatched method and returning to the same callsite, as illustrated in Fig. 10.

Another line of research on CFL-reachability focuses on its computational complexity. In general, the all-pairs CFL-reachability problem can be solved in $O(m^3n^3)$ time, where m is the size of its underlying CFL grammar and n is the number of its underlying graph nodes. Kodumal et. al [25] solve the Dyck-CFL-reachability more efficiently in $O(mn^3)$. Later, Chaudhuri [7] shows that the general CFL-reachability algorithm can be optimized into a subcubic one by exploiting the well-known Four Russians' Trick [26]. Recently, Zhang et. al [64] show that bidirected Dyck-CFL reachability can be solved in $O(n + p \ \log \ p)$ (where p is the number of its underlying graph edges) by noting that the reachability relation in a bidirected graph is an equivalence relation. This time complexity is further reduced to $O(p+n \cdot \alpha(n))$ in [6], where $\alpha(n)$ is the inverse Ackermann function. A few recent works focus on studying the complexity of interleaved Dyck-CFL reachability. Let $\mathcal{D}_k$ be a Dyck-CFL with k different kinds of parentheses. On a general graph, Reps et al [47] proved earlier that $\mathcal{D}_k \cap \mathcal{D}_k$ is undecidable when $k > 1$. Later, Englert et al. [10] proved that $\mathcal{D}_1 \cap \mathcal{D}_1$ is NL-complete. The complexity of $\mathcal{D}_k \cap \mathcal{D}_1$ remains open. On a bidirected graph, Li et al. [33] have recently shown that $\mathcal{D}_1 \cap \mathcal{D}_1$ is in PTIME and provides an algorithm computable in $O(n^7)$, which has subsequently been improved to $O(n^3 \cdot \alpha(n))$ by Kjelstrøm and Pavlogiannis [24]. In addition, Kjelstrøm and Pavlogiannis [24] also prove that $\mathcal{D}_k \cap \mathcal{D}_k$ is undecidable when $k > 1$. In this paper, the PAG for a program can be seen as a bidirected graph for L_C and L_R, but not as a bidirected graph for L_D since a reverse label $\overline{\ell} \in \{\overline{\textsf{new[t]}}, \overline{\textsf{dispatch[t]}}, \overline{\textsf{load[f]}}, \overline{\textsf{store[t]}}\}$ cannot be matched by ℓ in L_D. Therefore, L_{DCR} is undecidable (as discussed in Sec. 3.3). In addition, we have also introduced P3Ctx as an L_{DCR}-enabled pre-analysis that is linear in terms of the number of PAG edges in a program for accelerating kCFA without any precision loss.

For a CFL-reachability-based formulation [35, 37] proposed for supporting object-sensitive pointer analysis [40, 41] (denoted L^o_{FC} here), call graph construction is built-in naturally since object-sensitivity uses receiver objects as context elements. L^o_{FC} is defined as the intersection of two CFLs, $L^o_F \cap L^o_C$, where L^o_F ensures field-sensitivity as well as parameter passing and method returns at virtual callsites and L^o_C enforces object-sensitivity. Due to the nature of object-sensitivity [40, 41], L^o_{FC} adopts the allocation-site-based dispatch approach illustrated in Fig. 8c. At an allocation site, the type of the receiver object allocated can be deduced immediately (for supporting object-sensitivity), thus avoiding the need of encoding type information in the labels of some PAG edges such as new [t] and dispatch [t] as in L_D (for supporting callsite-based context-sensitivity). Thus, on-the-fly call graph construction is supported naturally in L^o_{FC} (even though the PAG of a program is still pre-built by applying a context-insensitive pointer analysis). For callsite-sensitivity, however, incorporating call graph construction into the traditional CFL-reachability formulation [50, 54, 63] is non-trivial (as described in Sec. 2). To the best of our knowledge, L_{DCR} represents the first such a solution.

5.2 Selective Context-Sensitivity

There are three types of approaches: (1) heuristic- or pattern-based [13, 30, 31], (2) data-driven [20, 22], and (3) CFL-reachability-guided [14, 15, 35–37]. By exploiting CFL-reachability, Eagle [35, 37], Turner [14, 15], and Conch [17, 19] represent recent efforts in accelerating object-sensitive pointer analysis [40, 41]. Selectx [36] represents the first attempt for accelerating kCFA with CFL-reachability. However, Selectx is not precision-preserving since its underlying L_{FC}-based formulation [54] does not incorporate a call graph construction mechanism. In this paper, we introduce P3Ctx, the first precision-preserving pre-analysis for accelerating kCFA, based on L_{DCR}.

6 Conclusion

We have introduced L_{DCR}, a new CFL-reachability formulation for supporting k-callsite-based context-sensitive pointer analysis (kCFA) with its own built-in call graph construction mechanism for handling dynamic dispatch. To demonstrate its utility, we have also introduced P3Ctx, which is developed based on L_{DCR}, for accelerating kCFA while preserving its precision. We hope that L_{DCR} can provide some new insights on understanding kCFA, especially its demand-driven incarnations [54, 55, 63], and developing new algorithmic solutions. In addition to selective context-sensitivity, we also plan to investigate the opportunities for leveraging L_{DCR} in library-code summarization [8, 50, 58] and information flow analysis [32, 39].

References

[1] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. Ph. D. Dissertation. University of Cophenhagen.

[2] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Association for Computing Machinery, New York, NY, USA, 324–341.

[3] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Mumbai, India) (POPL '15). Association for Computing Machinery, New York, NY, USA, 553566. https://doi.org/10.1145/2676726.2676977

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In Proceedings of the 33rd

International Conference on Software Engineering. IEEE, Honolulu, HI, USA, 241–250.

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. Association for Computing Machinery, New York, NY, USA, 243–262.

[6] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2017. Optimal Dyck reachability for data-dependence and alias analysis. Proceedings of the ACM on Programming Languages 2, POPL (2017), 1–30.

[7] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Association for Computing Machinery, New York, NY, USA, 159–169.

[8] Yifan Chen, Chenyang Yang, Xin Zhang, Yingfei Xiong, Hao Tang, Xiaoyin Wang, and Lu Zhang. 2021. Accelerating Program Analyses in Datalog by Merging Library Facts. In International Static Analysis Symposium. Springer International Publishing, Cham, 77–101.

[9] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In European Conference on Object-Oriented Programming. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 77–101.

[10] Matthias Englert, Ranko Lazić, and Patrick Totzke. 2016. Reachability in Two-Dimensional Unary Vector Addition Systems with States is NL-Complete. In Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16). Association for Computing Machinery, New York, NY, USA, 477484. https://doi.org/10.1145/2933575.2933577

[11] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-up context-sensitive pointer analysis for Java. In Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30-December 2, 2015, Proceedings. Springer International Publishing, Cham, 465–484.

[12] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. ACM Transactions on Programming Languages and Systems (TOPLAS) 23, 6 (2001), 685–746.

[13] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis. Association for Computing Machinery, New York, NY, USA, 1318.

[14] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:31.

[15] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2022. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. IEEE Transactions on Software Engineering (2022).

[16] Dongjie He, Jingbo Lu, and Jingling Xue. 2021. Context Debloating for Object-Sensitive Pointer Analysis. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, New York, NY, USA, 79–91.

[17] Dongjie He, Jingbo Lu, and Jingling Xue. 2021. Context Debloating for Object-Sensitive Pointer Analysis. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, New York, NY, USA, 79–91. https://doi.org/10.1109/ASE51524.2021.9678880

[18] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In 36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222), Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

[19] Dongjie He, Jingbo Lu, and Jingling Xue. 2023. IFDS-based Context Debloating for Object-Sensitive Pointer Analysis. ACM Transactions on Software Engineering and Methodology (2023).

[20] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–29.

[21] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs. Proc. ACM Program. Lang. 6, POPL, Article 58 (jan 2022), 29 pages. https://doi.org/10.1145/3498720

[22] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. Proceedings of the ACM on Programming Languages 1, OOPSLA (2017), 100.

[23] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 423434.

[24] Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The Decidability and Complexity of Interleaved Bidirected Dyck Reachability. Proc. ACM Program. Lang. 6, POPL, Article 12 (jan 2022), 26 pages. https://doi.org/10.1145/3498673

[25] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. ACM Sigplan Notices 39, 6 (2004), 207–218.

[26] VL Arlazarov EA Dinic MA Kronrod and IA Faradzev. 1970. On economic construction of the transitive closure of a directred graph. In Dokl. Acad. Nauk SSSR. 487–88.

[27] Michael John Latta. 1993. The intersection of context-free languages. Ph. D. Dissertation. USA. https://www.proquest.com/docview/304086568?pq-origsite=gscholar&fromopenview=true

[28] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In International Conference on Compiler Construction. Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169.

[29] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Transactions on Software Engineering and Methodology (TOSEM) 18, 1 (2008), 1–53.

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–29.

[31] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. ACM Transactions on Programming Languages and Systems 42, TOPLAS (2020), 1–40.

[32] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 780–793.

[33] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2021. On the complexity of bidirected interleaved Dyck-reachability. Proceedings of the ACM on Programming Languages 5, POPL (2021), 1–28.

[34] Leonard Y Liu and Peter Weiner. 1973. An infinite hierarchy of intersections of context-free languages. Mathematical systems theory 7 (1973), 185–192. https://doi.org/10.1007/BF01762237

[35] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. ACM Transactions on Software Engineering and Methodology (TOSEM) 30, 4 (2021), 1–46.

[36] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In International Static Analysis Symposium. Springer, Springer International Publishing, Cham, 261–285.

[37] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. Proceedings of the ACM on Programming Languages 3, OOPSLA (2019), 1–29.

[38] David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. Theoretical Computer Science 248, 1-2 (2000), 29–98.

[39] Ana Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–29.

[40] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis. Association for Computing Machinery, New York, NY, USA, 1–11.

[41] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology 14, 1 (2005), 1–41.

[42] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 308–319.

[43] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. Existential label flow inference via CFL reachability. In International Static Analysis Symposium. Springer, Springer Berlin Heidelberg, Berlin,

Heidelberg, 88–106.
[44] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 722–735.
[45] Jakob Rehof and Manuel Fähndrich. 2001. Type-based flow analysis: from polymorphic subtyping to CFL-reachability. ACM SIGPLAN Notices 36, 3 (2001), 54–66.
[46] Thomas Reps. 1998. Program analysis via graph reachability. Information and software technology 40, 11-12 (1998), 701–726.
[47] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. ACM Transactions on Programming Languages and Systems 22, 1 (2000), 162–186.
[48] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Association for Computing Machinery, New York, NY, USA, 49–61.
[49] Barbara G Ryder. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In International Conference on Compiler Construction. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 126–137.
[50] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In Proceedings of the Tenth International Symposium on Code Generation and Optimization. Association for Computing Machinery, New York, NY, USA, 264–274.
[51] Olin Grigsby Shivers. 1991. Control-flow analysis of higher-order languages or taming lambda. Ph. D. Dissertation. Carnegie Mellon University. CMU-CS-91-145.
[52] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Association for Computing Machinery, New York, NY, USA, 17–30.
[53] Manu Sridharan. 2007. Refinement-based program analysis tools. University of California, Berkeley.
[54] Manu Sridharan and Rastislav Bodík. 2006. Refinement-Based Context-Sensitive Points-to Analysis for Java. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 387400.
[55] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-to Analysis for Java. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Association for Computing Machinery, New York, NY, USA, 5976.
[56] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. ACM SIGPLAN Notices 35, 10 (2000), 264–280.
[57] T. Tan, Y. Li and J. Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 278–291.
[58] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for Computing Machinery, New York, NY, USA, 83–95.
[59] Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 263277.
[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In CASCON First Decade High Impact Papers. IBM Corp., USA, 214–224.
[61] WALA. 2022. WALA: T.J. Watson Libraries for Analysis. Retrieved April 14, 2022 from http://wala.sourceforge.net/
[62] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In European Conference on Object-Oriented Programming. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 98–122.

[63] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In Proceedings of the 2011 International Symposium on Software Testing and Analysis. Association for Computing Machinery, New York, NY, USA, 155–165.

[64] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. Association for Computing Machinery, New York, NY, USA, 435–446.

[65] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Association for Computing Machinery, New York, NY, USA, 197–208.